PROGRAM TRANSFERABILITY - DATA ACCESS REPRESENTATION FOR SECONDARY STORAGE

Stuart C. Schaffner, et al

Massachusetts Computer Associates, Incorporated

Prepared for:

Rome Air Development Center

November 1972

RADC-TR-72-289
Final Report
November 1972

PROGRAM TRANSFERABILITY - DATA ACCESS REPRESENTATION FOR SECONDARY STORAGE

Massachusetts Computer Associates, Inc.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, New York

169

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)

Applied Data Research, Inc. (Massachusetts Computer Associates)
Lakeside Office Park
Wakefield, MA 01888

2a. REPORT SECURITY CLASSIFICATION

UNCLASSIFIED

2b. GROUP

N/A

3. REPORT TITLE

PROGRAM TRANSFERABILITY - DATA ACCESS REPRESENTATION FOR SECONDARY STORAGE

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

FINAL

5. AUTHOR(S) (First name, middle initial, last name)

Stuart C. Schaffner
David B. Loveman
Robert E. Millstein

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
November 1972	158	7

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
F30602-71-C-0310	
b. PROJECT NO. Job Order No.: 45940202	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	RADC-TR-72-289

10. DISTRIBUTION STATEMENT

Approved for Public Release. Distribution Unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Rome Air Development Center (IRDA) Griffiss AFB, NY 13441

13. ABSTRACT

This report presents theoretical work which should lead fairly directly to analytical tools which can materially reduce the cost of transferring programs from one computer system to another. Past work has indicated that program transferability is a multifaceted problem requiring different solutions for different situations. This report concentrates on one such facet; namely, access to data stored on non-random access devices, such as tape and moving head disk. The report asserts that programs fail to be transferable in part because they either underspecify or over-specify their data processing requirements. A unified, general description of data files and data access methods, called the data access representation, is developed which, it is asserted, is detailed enough to allow efficient use of complex I/O devices, yet simple enough to make possible the development of analytical tools to study and modify programs using the data access representation. As an example of such a tool, an algorithm is developed which will alter a program to compensate for any of a class of data file structure transformations similar to those required to transfer a data file from one I/O device to another. The data management routines of three important operating systems are then considered: IBM OS/360, CDC SCOPE, and HONEYWELL GECOS. Each is described in detail in terms of the data access representation.

DD FORM 1 NOV 65 1473

UNCLASSIFIED

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Data access methods						
Data access representation						
Transferable programming						
Transferable environment						

PROGRAM TRANSFERABILITY - DATA

ACCESS REPRESENTATION FOR SECONDARY STORAGE

Stuart C. Schaffner
David B. Loveman
Robert E. Millstein

Massachusetts Computer Associates, Inc.
(Formerly Applied Data Research, Inc.)

FOREWORD

This technical report was prepared by Massachusetts Computer Associates, formerly Applied Data Research, Inc., Lakeside Office Park, Wakefield, MA, 01888, under Contract No. F30602-71-C-0310, Job Order No. 45940202. The report has been reviewed by the Office of Information, RADC, and has been approved for release to the National Technical Information Service (NTIS).

The RADC project engineer was James A. McNeely (IRDA).

This report has been reviewed and is approved.

Approved:

JAMES A. McNEELY
Project Engineer

Approved:

FRANZ H. DETTMER, Colonel, USAF
Chief, Intel and Recon Division

FOR THE COMMANDER:

FRED I. DIAMOND
Chief, Plans Office

ABSTRACT

This paper presents theoretical work which should lead fairly directly to analytical tools which can materially reduce the cost of transferring programs from one computer system to another. Past work has indicated that program transferability is a multifaceted problem requiring different solutions for different situations. This paper concentrates on one such facet, namely, access to data stored on non-random access devices such as tape and moving head disk. The paper asserts that programs fail to be transferable in part because they either underspecify or overspecify their data processing requirements. A unified, general description of data files and data access methods, called the data access representation, is developed which, it is asserted, is detailed enough to allow efficient use of complex I/O devices yet simple enough to make possible the development of analytical tools to study and modify programs using the data access representation. As an example of such a tool, an algorithm is developed which will alter a program to compensate for any of a class of data file structure transformations similar to those required to transfer a data file from one I/O device to another. The data management routines of three important operating systems are then considered: IBM OS/360, CDC SCOPE, and HONEYWELL GCOS. Each is described in detail in terms of the data access representation.

TABLE OF CONTENTS

1.	PROGRAM TRANSFERABILITY - A REVIEW		1
2.	ON THE NEED FOR DEVICE-DEPENDENT CODE		6
3.	PHYSICAL REPRESENTATION		10
	3.1	Volume	10
	3.2	Device	10
	3.3	Graphical Representation	11
	3.4	Tape: An Example of a Physical Representation	15
4.	DATA ACCESS REPRESENTATION		18
	4.1	Level of Detail	18
	4.2	The Data Access Technique	19
5.	TRANSFERABILITY OF PROGRAMS USING THE DATA ACCESS REPRESENTATION		32
	5.1	Characterization of Programs	32
	5.2	Characterization of Program Execution	39
	5.3	Associate Graph Transformations	44
6.	ADDITIONAL MACHINERY FOR THE DATA ACCESS REPRESENTATION		64

TABLE OF CONTENTS (Con't.)

6.1	Modification of Data Structures	64
6.2	Representative Nodes	70
6.3	Choice Brackets	72
6.4	The (?) Node	74
7.	IBM OS/360 ACCESS METHODS	76
7.1	Introduction	76
7.2	Access Methods	76
7.3	Sequential Access Methods	77
7.4	Partitioned Access Methods	92
7.5	Indexed Sequential Access Methods	99
8.	CDC SCOPE ACCESS METHODS	109
8.1	SCOPE Access Method Elements	110
8.2	Forward Sequential File Structure	116
8.3	Doubly Sequential File Structure	116
8.4	Random Access File Structure	118
8.5	SCOPE Abstract Machine	120
8.6	SCOPE Data Access Macros Useable on Sequential Files	125

TABLE OF CONTENTS (Con't.)

	8.7	Macros and Transforms Useable on Random Access Files	134
	8.8	File Structure Templates	137
9.	HONEYWELL (GE) 600 GEFRC (GENERAL FILE AND RECORD CONTROL)		139
	9.1	Introduction	139
	9.2	File Structure -- Standard System Format	139
	9.3	File Control Block	144
	9.4	Buffering	147
	9.5	Logical Record Processing	148
	9.6	Device Positioning Commands	154
	9.7	Physical Record Processing	156
	9.8	Input/Output Editor Functions	156
	9.9	File Preparation Commands	158

BIBLIOGRAPHY

EVALUATION

Contract F30602-71-C-0310, Environment for Transferability, was undertaken to investigate the possibility of minimizing the propagation of hardware intelligence into intelligence application programs operating on them.

Currently, Data Management Systems are not transferable among machines of different manufacturer's computer systems in spite of being programmed in the same language.

Systems of the size and complexity needed to provide adequate support to intelligence functions must be tailored to the capabilities of a specific manufacturer's hardware. When the users of these computer systems (hardware/software) are forced to change to a different computing environment, supplied by a different manufacturer, they must redesign all their application programs. This is very expensive.

In an attempt to rectify this condition (Ref RADC TOP 4) RADC decided to conduct an investigation of the characteristics of the data access techniques used with different computer manufacturers. The investigation included a detailed examination of the different data management systems and resulted in the development of an algorithm which can be used to alter a program to compensate for a class of data file structure transformations similar to those required to transfer a data file from one I/O device to another.

The effort provides the necessary tools to support the design, development/modification, and implementation of transparent computer software for use within the intelligence community.

JAMES A. MCREELY
Project Engineer

1. PROGRAM TRANSFERABILITY - A REVIEW

It is appropriate to begin this paper with a review of our previous work, which constitutes the foundation of our current effort. The first point to note is our definition of the program transferability problem. In order to reduce an amorphous, probably unsolvable problem to manageable proportions we imposed several conditions:

- We are concerned with the transfer of large programs -- written in a high level language; consisting of many pieces interconnected in a more or less complicated way; interacting with secondary storage.

- Although the difficulties associated with standardization are real - and we will, in fact, introduce new suggestions for standardization (at least functionally) - these difficulties are mainly administrative, not technical, and will not concern us here.

- We restrict the computing milieux among which transferability is feasible to machines of similar design and similar capacity -- that is, "FORTRAN machines" with comparable memory sizes. We will consider transferability between a 360/65 and an 1108, but not between a PDP-8 and a 6600.

- We regard the transferability problem as solved when a program running with acceptable efficiency on one machine can be moved, at acceptable cost, to another machine on which it again runs with acceptable efficiency. This viewpoint implies that such a solution will not allow the "last

inch" of operating efficiency to be obtained in a program that is to be transferable. We accept this loss of efficiency as the price we pay for moderate transferability costs. As in so many other areas of computer science, this position is a balance between two competing demands.

We shall see in the sequel why we introduced these restrictions, but let us leave them now.

We regard a program as consisting of three parts

- Algorithm - the core of a program is the algorithm which is to be implemented. The point of the third restriction above is that the algorithm implemented is sensitive to computer design and capacity. One would use different sorting algorithms on a PDP-8 and an ILLIAC IV. In order to have any basis at all for accomplishing transferability, we had to have a constant, and we chose it to be the algorithm designed to solve a problem rather than the problem itself. The algorithm is conventionally described in a high level (algebraic) language - e.g., FORTRAN, ALGOL - and we repeat that we will not consider the very real problems of standardization of such languages.

- Program assembly - large programs are commonly written in pieces which have to be assembled into running modules. The pieces can be code or data and can be related as subroutines, coroutines, overlays, job steps, etc. We call the process of gluing these pieces together program assembly

- File structures - the programs we are concerned with interact with secondary storage They are sufficiently large and complex that all code and data cannot be core contained, and hence they require a file structure to manipulate objects in secondary storage.

Now, perhaps the most transferable program is a simple FORTRAN main program without subprogram calls and without I/O. This suggests that the difficult transferability problems lie in the areas of program assembly and file structures. We first note that these two troublesome program parts interact more with operating system than hardware features -- e.g., with loaders and file handlers rather than with arithmetic units and memory address registers. This implies that functional standardization of at least some operating system features might be necessary to effect program transferability.

High level programming languages provide an effective means of describing an algorithm (and, hence, standardization of these languages could be expected to solve the transferability problem for at least this portion of a program). When we consider program assembly and file structures the problem is not so simple. We do not have high level problem-oriented languages for describing these parts of programs. The languages (e.g., JCL) provided not only must describe the logical (problem-oriented, algorithm-determined) characteristics of the desired program assembly and file structures, they must also describe, and be couched in terms of, the physical (machine-oriented) mapping of these program parts into the operating system-hardware complex. This mapping is a series of calls on operating system and hardware capabilities. This dual nature of the languages describing program assembly and file structures lies, we believe, at the heart of the transferability problem. When a programmer describes these program parts

he serves two masters - one, the algorithm he is implementing (and, ultimately, the problem he is solving) and, two, the hardware-operating system on which he is implementing the algorithm. Unfortunately, he has only one language to serve these two purposes. He must describe both the logical structure and its mapping to physical realization with only one tool.

Now the portion of this description which is algorithm dependent is transferable, but the remaining, machine-dependent, part is certainly not. Suppose a programmer could write his entire program - algorithm, program assembly, and file structures - in two colors, so that the statements in black are algorithm dependent and hence transferable, but the statements in red describe the mapping of the algorithm dependent parts into a particular machine. Then transferable programming would consist of describing a problem solution in black and supplying additional red statements for every computer facility on which one wished to realize that problem solution.

Unfortunately, programmers do not have two colors available to write in. We believe that the present difficulty with transferring programs arises because of this. Present facilities submerge the algorithm-dependent portions of a program in a mass of mapping description. It is necessary to work backward from the realization of an algorithm to reobtain the logical structure of the algorithm, program assembly, and file structures. Our proposal to solve the transferability problem is to provide separate means of describing the algorithm-dependent and machine-dependent portions of a program.

Let us now examine the current situation in terms of the availability of black and red languages.

- Algorithms - high level algebraic languages provide suitable black languages. Red languages, which would describe the mapping from a black language to machine language are not necessary because we already have software (compilers and interpreters) which perform this mapping.

- Program assembly - it is possible to describe a (hopefully) sufficient black language in terms of the various possible logical relationships (subroutine, coroutine, etc.) among program pieces. A suitable red language would describe the physical relationship (overlays, etc.) among these same pieces. The problem is more fully discussed in the "Handbook on File Structuring" and "The Representation of Algorithms".

- File structures - this problem is somewhat more difficult, if only because of the great diversity of secondary storage devices. Our current effort is centered around obtaining functional descriptions of meta-devices which are sufficiently general that they provide a framework in which logical file descriptions can be made. This would constitute a black language. In addition, these functional descriptions must be clearly mappable into a large range of physical devices. Such a mapping would constitute a red language. The remainder of this paper will describe our efforts to obtain such descriptions.

2. ON THE NEED FOR DEVICE-DEPENDENT CODE

One of the most direct ways of ensuring "transferability" of a user program is to write it for a device-independent environment. The operating system then contains mapping software which will support this general environment on any of a range of specific machine configurations. This user program may then be transferred without change between any two machine configurations within the range of the mapping software of the operating system.

This method has been used with success for efficient random-access storage devices such as fixed-head disk and drum. The device-independent environment includes a virtual memory space within which the user program may directly store and access the data it uses. This virtual memory space is partitioned by the operating system into pages or segments, which are mapped onto blocks of memory on the random access devices. A reference to a particular location in virtual memory is mapped automatically into a reference to the corresponding block of memory on the random access device.

In order to efficiently use a virtual memory environment the user program must organize its virtual memory accesses to minimize the size of, and number of changes to, the working set of device memory blocks. Techniques for accomplishing this are fairly well understood and are fairly independent of the specific devices upon which the virtual memory is mapped. Thus a program written for such a system will be truly transferable as we have defined the term; the program will be reasonably efficient in its original form, will require practically no reprogramming for a new computer configuration, and will run reasonably efficiently on the new configuration.

Random access devices, however, are not suitable for all types of secondary storage. To provide fast access to any point in memory no matter what point was accessed before requires complex and expensive equipment. For example, the cost of 256K of fixed head disk memory for a PDP-11/20 computer with 8K of core is roughly equal to the cost of the rest of the system.

Th re is another class of storage devices, however, which offers greatly reduced cost per word of memory but which allows efficient access to that memory only in certain sequences. We shall call these devices moving head devices. The most common examples are tape, moving head disk, and data cell. These devices are especially useful when there is a large amount of data which is accessed in some particular sequence, for instance, while sorting data or while updating an information data base.

Data stored on such a device can be thought of as having two simultaneous structures, physical structure and access structure. The physical structure is engendered by the fact that the data is stored on a physical object, and thus a word can be said to have a definite position in real space at any given time. Thus a particular word of data may be thought of for instance, as the 3rd word of the 5th track of the 2nd cylinder of the disk pack A0001. As we shall see in part 3 of this paper, the physical structure representation of data is not only highly sensitive to alterations in machine environment but also not really satisfactory as a data representation even within one environment.

Data stored on a moving head device is also structured by the set of head positioning commands allowable on that device. We call this the access structure of the data. This structure is of greater use to the programmer, as it contains explicitly the information he needs to effectively access data in

some particular sequence. It is the access structure of data which forms the basis for data management systems such as the IBM OS/360 data access methods. Data on tape is "sequential" not because words appear sequentially on the oxide layer of the tape but because the tape head passes over these words in one particular sequence when the tape drive is instructed to read or write forward. We shall discuss this in more detail in part 4 of this paper.

The access structure of data comes, then, partly from the physical arrangement of data words on a storage volume and partly from the set of possible sequences in which these words may be accessed by the device upon which the volume is mounted. Thus, for instance, a tape mounted on a tape drive which reads or writes forward and backward has a radically different access structure than the same tape mounted on a tape drive that reads and writes only forward.

Can one solve the problem of transferability for moving head devices by developing a completely machine independent programming environment, as has been done successfully for random access devices? We feel not, at least for the near future. It is, to be sure, fairly easy to map a particular physical structure representation of some data on one device onto an "equivalent" physical structure representation on some other device. If these two devices have different access capabilities, however, the access structures of these two "equivalent" data representations may differ significantly.

A tape rewind may take several seconds; a disk head seek may take several hundred milliseconds; a typical CPU instruction, however, takes only a few micro-seconds. This great disparity in speed between the algorithmic and data access parts of a program using moving head devices

usually implies that data access efficiency rather than computational efficiency determines the overall program efficiency. In other words, using an algorithm which "wastes" several hundred CPU instructions in order to save one disk head seek or tape rewind is usually a very good trade-off.

We must always keep in mind that the ultimate purpose of a user program is to solve some problem for that user. The algorithmic, program assembly, and data access parts of a program are simply means to that end. This is why we include in the cost of transferability of a user program not only the cost of alteration but also the cost of using a suboptimal problem solution.

It may eventually become possible for a user merely to state the problem he wishes solved. System software will select an algorithm and data access scheme suitable for the particular machine environment that obtains. These problem statements will be truly machine independent.

Such a system does not appear feasible in the near future, at least for data management problems. Until it does become feasible it is futile to attempt to solve the problem of transferability for moving head devices by creating machine independent languages. Rather, we should strive to reduce the machine dependence of the languages to that minimum necessary to utilize the special characteristics of a given environment. We must then develop analytical tools to reduce as much as possible the cost of altering this machine dependent code when the program is to be transferred to a different machine environment. We begin to do this in part 5 of this paper.

3. PHYSICAL REPRESENTATION

It is possible to represent access to secondary storage by describing in detail the physical layout of the storage volume and the mechanical actions of the I-O device upon which the volume resides. We shall call this the physical representation of secondary storage access.

3.1 Volume

A volume is some physical entity such as a tape or a disk which is capable of storing data. We shall assume that this data is broken up into units, all of the same size, which we shall call words. We shall further assume that each word has a precise and unchanging position on the volume and that any control datum, such as an interrecord gap on tape, consists of some integral number of words. These assumptions aren't always true, as we shall explain later.

3.2 Device

The volume is placed on some machine called an I-O device. All I-O devices commonly used for secondary storage are basically similar and may all be described reasonably well by the following model.

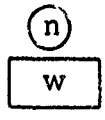
There are two parts to the model device, a volume positioner and a read-write head. The volume positioner orients the volume so that the read-write head is always "at" exactly one word. The head may be given a command to read or write the word it is currently at, or both. Associated with the word, however, is an access protection attribute which may make reading or writing (or both) of that word illegal. The volume positioner may be given one of a set of commands. Any of these commands will cause the positioner to move the volume so that the head is over another word. The current head

position and the command name determine exactly which word is selected. The time required to move the volume is also determined by these two parameters.

The fact that the pair (current head position, position command) defines a unique next word implies that the device has no memory of its past actions. This is an important simplification, one that we would like to retain if at all possible. It is in the main true, but there are exceptions.

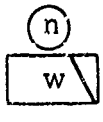
3.3 Graphical Representation

We may express this model graphically. We shall denote a word by

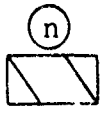


where n is the name of the word, and w its contents. If the word is read-only we denote it by

and if it is write-only by

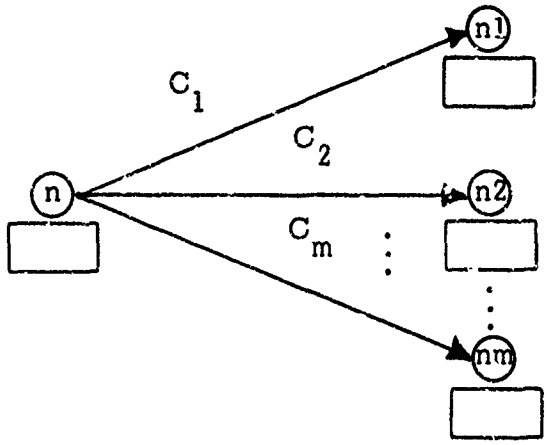


If it is not readable or writable we denote it as



or simply

For every volume positioning command that is legal for a given word we draw an arc from that word to the word selected by that command. We label the arc with the name of the command it represents.



Once we have included all the words on the volume and have drawn for each all legal command arcs we have a graph which represents the model. We shall often call the words nodes. We shall at times call the word currently under the read-write head the state. We represent this state by a token, which is "on" exactly one node at any given time. We call the graph the volume access graph.

Let us now state some properties of a volume access graph. First the graph is an s-graph; that is, there may be more than one arc from a given node to a given other node. This corresponds to the possibility that when the head is at certain words on the volume two or more commands may have the same effect. Second, the number of arcs emanating from a given node is bounded by the number of commands recognized by the device. For all devices currently used this bound is finite. Third, the number of nodes is finite. This corresponds to the fact that each node represents a unique area on a volume of finite physical dimensions. Properties two and three together

guarantee that the graph is finite. Fourth, there is at most one arc with any given label emanating from any given node. This corresponds to the fact that the device operates deterministically.

The fifth, and most interesting, property is that this graph is strongly connected for almost all volumes of interest. We shall show that if a volume graph is not strongly connected then it represents only a part of a complete process of information storage and retrieval. We make two assumptions:

1. Let G be the set of all nodes in the volume graph. There is a nonempty set $S \subseteq G$ of nodes called starting states for which

   $\forall n \in S$ then

   $\forall m \in G \ \exists$ a path $p: n \to m$

2. For every $m \in G$, $\exists$ a path q and a node $\ell \in S$ such that $q: m \to \ell$

From these two assumptions it is easy to prove strong connectedness. We must prove that

$$\forall m, n \in G \quad \text{there is a path } p: m \to n$$

By assumption two, there is a path $q_1: m \to \ell$. for some $\ell \in S$. By assumption one, there is a path $q_2: \ell \to n$. But then the path $q_2 q_1 : m \to n$ and thus $p = q_2 q_1$. q.e.d.

Let us examine the motivation for these assumptions. Assumption one implies that it is possible to initialize the volume and the device such

that all parts of the volume can be accessed, no matter what the past history of this volume has been. If the assumption one were not true, then it would be possible to make parts of a volume permanently inaccessible on that device. Assumption two implies that it is always possible to re-initialize the volume. Let us consider several important devices:

1. Tape

   S contains the first word on the tape. Assumption two can be met by rewinding the tape. Assumption one is met by loading the tape.

2. Disk

   Here S = G and the spinning of the disk satisfies both assumptions.

3. Card punch

   A card punch does not necessarily meet assumption two. If the punch does not accept already punched cards in its input hopper then it will not be possible to reinitialize a partially punched deck of cards.

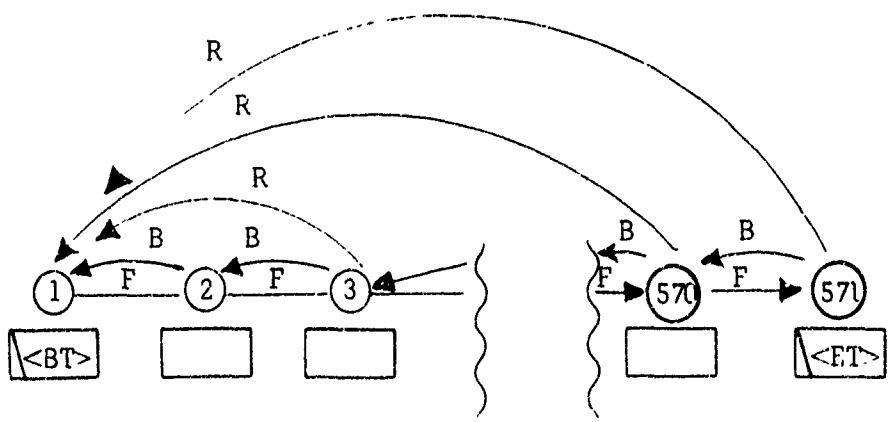
Assumptions one and two were created on the premise that data which is read must first be written and data which is written will presumably eventually be read. The reason card punches and line printers fail to meet these assumptions is that the subsequent readings are performed by different devices, namely card readers and humans, respectively. Obviously, a human will access a printout differently than a line printer, and the processes must be described by different graphs. We will call any device which produces a graph which is not strongly connected an incomplete device. It

should be stressed that incomplete devices are inherently less flexible than complete ones. We shall be able to describe accurately the use of an incomplete device, but we shall not be able to use some of the transformations and simplifications that we shall derive for complete devices.

3.4 Tape: An Example of a Physical Representation

While we won't attempt here to completely describe a real tape machine, we will concoct a simplified "tape" machine which still contains some of the interesting features of a real one.

Our tape has 571 words on it. Each word may contain an integer with absolute value less than 2^{15}, or one of the special codes <RG>, <BT>, and <ET>. The first word on the tape contains <BT> and is read-only. The 571^{st} word contains <ET> and is also read-only. The rest of the tape contains data and record gaps. A record gap consists of at least 4 words in succession all containing <RG>. The physical volume graph is as follows:



Our three volume positioning commands are then F, B, and R standing for space forward, space backward, and rewind, respectively. A volume positioning command is of the form

MOVEH α

where α = F, B, or R

The current word may be accessed by three different commands:

1. READW
2. WRITEW
3. ON NODE (C1, C2, ..., CN) GO TO (S1, S2, ..., SN)

   where

   Ci $(1 \leq i \leq N)$ can be DATA, <RG>, <BT>, <ET>
   Ci $\neq$ Cj for $i \neq j$
   Si $(1 \leq i \leq N)$ is a statement label

READW causes the current word to be read and its contents placed in some data transfer register. WRITEW causes the contents of some data transfer register to be written into the current word. ON NODE causes the program to branch to statement Si if the current word is of type Ci .

We now have a representation of a tape and a tape drive, and a notation which allows us to write programs for them. Unfortunately, the atomic operations F, B, and R are simply not realizable on a standard tape drive. A tape drive takes some time to start and stop a tape, and during this time more than one word will pass by the read-write head. Consequently, the simplest tape commands generally deal with entire records. The set of tape record commands do <u>not</u> strongly connect the data graph.

As an example of a real tape drive command let us represent READER. READER assumes that the tape is stopped with the head in the interrecord gap. The tape is started and two words pass by the head before the tape reaches speed. READER then ignores any further <RG> words. When it encounters DATA words it reads them. After reading at least one DATA word, it begins checking for an <RG> word. When it encounters one it stops reading and halts the tape. As the tape slows down, one word passes by the head. We may express READER as follows:

```
        MOVEH       F
S2      MOVEH       F
        ON NODE     (DATA, <RG>) GOTO (S1, S2)
S1      READW
        MOVEH       F
        ON NODE     (<RG>, DATA) GOTO (S3, S1)
S3      MOVEH       F
```

By describing tape operations in such minute detail, we have introduced another problem. At the READER level the tape drive always performs consistently. At the F, B, and R level it does not. All interrecord gaps are not exactly the same size, nor are they generally exactly an integral multiple of data words in width. The drive does not stop a moving tape in some precise distance, it only stops it within some range of distances. Thus our graph represents a much neater and more consistent situation than really obtains.

4. DATA ACCESS REPRESENTATION

4.1 Level of Detail

The physical volume representation, while it has many attractive features, is not suitable as a language for expressing considerations of transferability. Its principal inadequacy is that it expresses secondary storage operations in excessive detail. Thus instead of illuminating the important characteristics of a data volume, it obscures them in a mass of irrelevant detail. We were forced to go to this level of detail by two restrictions:

1. We required that each node in the volume access graph correspond to a precise physical position on the volume.

2. We required that the access graph state always correspond to the precise position of the physical read-write head.

If a user program is to interface directly with an I-O device without any intervening software then these restrictions are necessary. It is generally accepted, however, that a program which contains machine-level instructions for an I-O device is seldom transferable. We shall always assume that transferable programs communicate with secondary storage through some standardized data management routines. These routines hide minor differences between similar devices and insulate the user program from real-time constraints imposed by machine dynamics.

Given that we may interpose a data management routine between the user program and the device, we have great freedom to choose to what degree the details of device handling are left to the user and to what degree they are

handled automatically. At one extreme is the volume access graph, which we have already rejected as being too detailed. At the other extreme the strong connectedness of most of our graphs allows us to construct the trivial case of a complete, or direct-access graph where it is possible to get from any node to any other node in one step. Programs written for such a data management routine would be completely "transferable" in that they could run with no modification on almost any machine and almost any operating system. They would not be transferable in our sense of the word, however, in that it would be almost impossible to assign meaningful costs to the arcs of the graph. Thus it would be quite difficult to optimize a program to efficiently use the special capabilities and avoid the special limitations of a particular device.

This discussion should make it clear why we included in our definition of transferability both the cost of recoding a program for a new environment and the increased cost of running the recoded program in that environment. The volume access graph level allows a programmer to minimize the run-time costs but at the expense of losing all control over the reprogramming costs. The complete-graph approach, however, reduces the reprogramming cost to near zero but at the expense of losing all control over run-time costs. We need an intermediate level which is detailed enough to allow reasonable control over specific devices yet general enough to allow analytical techniques to be applied to the process of reprogramming.

4.2 The Data Access Technique

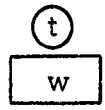
We will now develop a language intermediate in level of detail between the physical volume representation and the complete-graph representation. We shall retain the graph structure developed for the physical volume representation, but we shall weaken its correlation with specific locations on a

physical volume. We shall retain the notion of a device acting upon the graph through a finite command set. Our device, however, will be not a physical device but a virtual device formed by interposing a data management program between the user program and the device.

4.2.1 Nodes

As before, the basic unit of data will be the word, and there will be one word per node of the graph. We shall no longer require that a node correspond to a fixed position on a storage volume, but we shall require that it have a fixed logical relationship to all other nodes in the graph. We shall describe this logical relationship when we discuss templates. As before, each node shall have a name. We shall, however, call it the node type instead of the node name, and we shall no longer require that it be unique within the graph. We shall discuss this wher we examine the problem of context.

As before, the symbol for a node shall be



where w is the data word and t the node type.

4.2.2 Arcs

As before, the basic changes of state within the graph are described by arcs. Each arc has associated with it a label drawn from a finite set of labels called the positional command set. Now, however, an arc represents a change of state for the data management routine and device together, rather than some definite physical movement. We shall again

associate with each arc a cost. This cost shall now, however, be a mean cost as the operation the arc stands for no longer necessarily represents a single physical action. If we have chosen our positional command set with reasonable care the deviation from this mean will not be unworkably large.

4.2.3 Templates

One of the objections to the tape graph developed earlier was that the number of data words on a particular tape was not constant. It varied according to the number of interrecord gaps, the amount of friction in the capstan brake, the temperature, and so forth. Thus a permanent representation of a particular tape as a single graph was not logically consistent. Even if we had a nonstretchable tape and a tape drive with inertialess moving parts, it could be true that neither the user nor the data management routines know how many words are on the tape until the tape is completely read or written. We must find a precise representation for statements such as "This is a tape, but of unknown length".

Such a tape is describable by a set of graphs, one for each possible tape length. The set is enumerated by a single parameter, namely tape length. This parameter has a lower and an upper bound. We may specify the value of the parameter to be any value between the lower and upper bounds. Specifying the parameter results in a new set of graphs that is a subset of the old set. We call such a set of graphs a template.

We shall not define templates in general. Our use of templates is at present rather limited and we do not know yet precisely what attributes a template should or should not have. We shall content ourselves with a limited operational definition which we shall augment as the need arises.

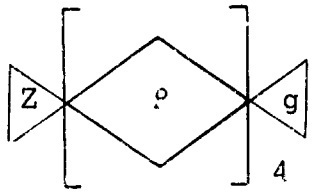
The basic construct we shall use to build templates is the iteration bracket. Its basic form is as follows:

$$LP\ [\ P\ ]_i\ RP$$

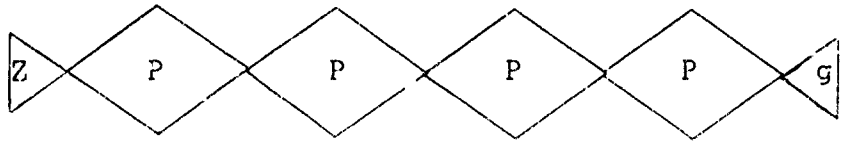
where LP, P, and RP are pictures

i is a non-negative integer

The position of the left and right brackets, which are parallel, the same height, and at the same level, defines a set of boundaries for LP, P, and RP. We say that LP has a right edge, P has both right and left edges, and RP has a left edge. We can construct a picture by matching appropriate edges. The basic form given above corresponds to a single picture formed from LP, RP, and i versions of P. The i versions of P are chained together by placing the left edge of the second P, if it exists, against the right edge of the first P, and so on. The right edge of LP is then abutted with the left edge of the first P and the left edge of RP with the right edge of the i-th P. If i = 0 the right edge of LP is abutted with the left edge of RP. Thus



stands for



We now introduce uncertainty into templates by allowing

$$LP\ [\ P\ ]_{q=i}^{j}\ RP$$

where

q is a parameter name

i and j are integers such that $i \leq j$

This stands for all of the pictures obtainable by iterating P at least i times and at most j times. It should be clear that

$$LP\ [\ P\ ]_{3}\ RP$$

and

$$LP\ [\ P\ ]_{q=3}^{3}\ RP$$

produce the same picture. We describe the process of selecting one of the possible pictures by assigning to the parameter name q an integer value such that $i \leq q \leq j$. We shall allow infinity, ∞, as a possible value for j. We shall abbreviate

$$[\ ]_{q=0}^{\infty} \quad \text{as} \quad [\ ]_{q}$$

At times we would like to differentiate between elements of an iteration. We shall therefore allow the following

$$[\ *p\ ]_{p}$$

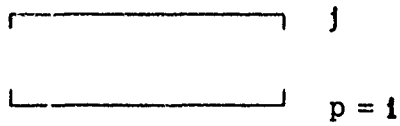
where *p appears somewhere inside the picture. *p will be replaced by 1 in the first element of the iteration, by 2 in the second, and so forth up to p. Thus

$$A\ [\ L\ *p\ ]_{p=3}\ B$$

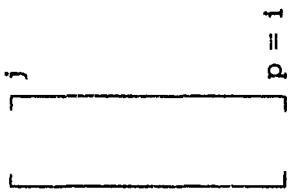
stands for

A L1 L2 L3 B

We shall allow one further type of iteration bracket:



which is equivalent to



4.2.4 Macros

The structure of an iteration bracket

$$LP\ [\ P\ ]_{q}\ RP$$

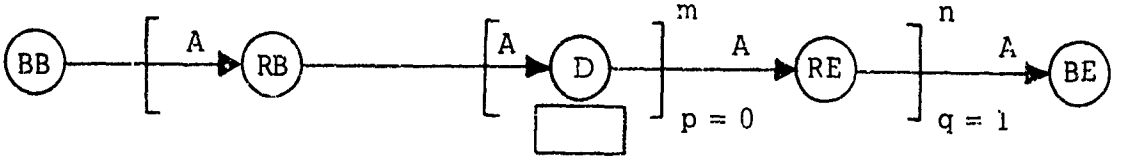
gives templates constructed using brackets a particular form. LP and RP are invariant outer parts. The chain of pictures P is enclosed by LP and RP. Its variation is completely described by the parameter q.

Thus it is often true that a template consists of a fixed number of external nodes and a variable inner structure with the variation controllable by a fixed number of parameters. We call a template such as this a macro. For each macro we assign a unique graphic symbol. The parameters are assigned unique locations within the symbol and the external nodes are assigned unique locations on the periphery of the symbol.

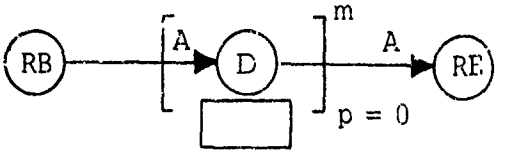
We may use any macro so defined as part of a larger template by placing arcs between the proper points on the macro's periphery and other nodes or macros. In fact, we may even define a new macro using the template so constructed.

We shall not require that all iteration brackets be given explicit parameter names, nor that all parameters within a macro appear on the macro symbol. Thus it may be possible to specify all known parameters of a template and still have a template containing more than one graph. We say that such templates have <u>implicit parameters</u>.

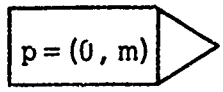
As an example, consider the following template



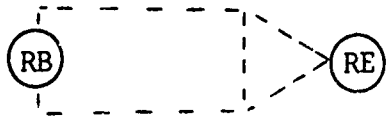
We may replace



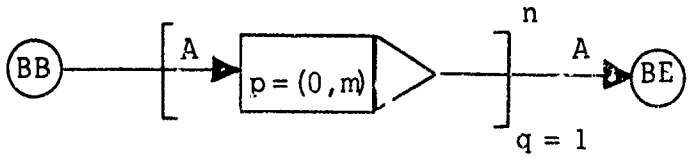
by the macro



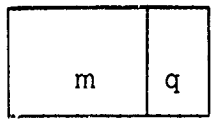
with equivalence



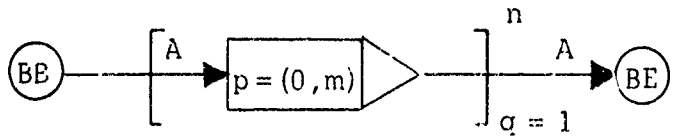
Thus the template is now



We could further simplify the template by defining the macro

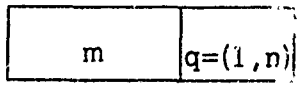


which replaces

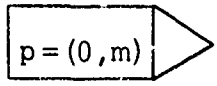


and with equivalence

Our template has now been reduced to



Note that specifying the values for p in



results in a single graph, while specifying values for m and n in

results in a template containing several graphs.

4.2.5 Template Names

It will be necessary later to describe a template compactly through use of some standard format. To accomplish this we shall define a template name as follows:

$$
\begin{aligned}
&\langle \text{TNAME} \rangle && ::= \langle \text{MACNAME} \rangle \\
&&& \text{or} \quad (\langle \text{TNAME} \rangle) , \langle \text{PLIST} \rangle \\
&\langle \text{PLIST} \rangle && ::= \langle \text{PRMS} \rangle \{ , \langle \text{PRMS} \rangle \}_0^{\infty} \\
&\langle \text{PRMS} \rangle && ::= \langle \text{PNAM} \rangle = \langle \text{BNDS} \rangle
\end{aligned}
$$

where

$\langle \text{MACNAME} \rangle$	=	name of a macro
$\langle \text{PNAM} \rangle$	=	name of a parameter for some template
$\langle \text{BNDS} \rangle$	=	limits on the value of the parameter

We shall assume that all templates have been described by a set of standard macros or are refinements of these macros. Each macro in this set has been given a name.

As we described earlier, a template is refined by specifying values for, or value limits on, one or more of the explicit parameters of the template. These values or value limits must be within the value limits already in effect for that template. The result is a new template which is a subset of the old. This new template may have some explicit parameters which were implicit in the old. The process of refinement may be carried further by specifying parameters of the new template, and so forth. In the definition of template name, $\langle \text{PLIST} \rangle$ is a list of parameter specifications sufficient to carry out one level of refinement. Parentheses delimit multiple levels of refinement, e.g.

$$(((\langle \text{TNAME} \rangle) , \langle \text{PLIST} \rangle) , \langle \text{PLIST} \rangle) , \langle \text{PLIST} \rangle$$

represents three levels of refinement of $\langle \text{TNAME} \rangle$.

4.2.6 Access Technique State

In the data access representation the physical read-write head is replaced by a virtual head. The state of the system represents not the precise position of a physical entity, but rather a collection of information maintained by a data management routine in order to properly manage a storage volume. Some of this information may be stored on the volume itself, such as record marks and home addresses. The rest of the information will be contained in pointers, counts, and flags internal to the data management routine.

It is crucial to program transferability that the information a program may obtain about secondary storage be rigorously defined and strictly limited. It must be possible at any point in a user program to be able to tell exactly how much that program could know about the data structure it is accessing. We call this information the context.

The user program gains context information from two sources. The control portion of the program, in setting up the data file and initializing the data access technique, establishes an initial template. Once the algorithmic part of the user program is given control, it may gain further structural information about the data it is accessing by using the command

ON NODE (T1, T2, ..., Tn) GOTO (S1, S2, ..., Sn)

where

T1, T2, ..., Tn are node types, no two of which are identical

S1, S2, ..., Sn are program statement labels

Tn may be < ELSE > , which stands for all node types not specified in T1, T2, ..., Tn-1. If the current node is of type Ti then control will pass to statement Si . Any statement label may be replaced by < NEXT > , which stands for the statement immediately following the ON NODE statement. If no Ti matches the current node type then the statement is in error.

4.2.7 Head Movement Commands

At any given point in a user program the data access technique is in a definite state, represented by a node in the current template. From this node emanates at least one arc to another node. Each such arc has a label, and no two arcs from the same node may have the same label. At any point in the user program the command

MOVEH <ARC>

may be given, where <ARC> is an arc label. If the current node has no such arc emanating from it the statement is in error. If there is such an arc then upon return the node terminating the arc will be the new current state.

Note that the program is not allowed to test the current state to determine the labels of arcs emanating from it. This is quite important. If we did allow such a test, then we would be classifying a node not just by its type t, but by the name (t, A), where A is the (unordered) set of names of arcs emanating from it. This is a more complex system then is desired or needed.

4.2.8 Data Access Commands

If the current node has associated with it a readable or writable data word then that word may be read or written, respectively. We shall later use some very specialized commands to read and write using buffers. For our present purposes, however, we may define the access technique's capability by the two commands

READW and WRITEW

We assume some data transfer register R exists. READW will, if the current node is readable, set R to the data value of the current node. WRITEW will, if the current node is writable, set the data value of the current node to the value in R.

For many devices it is not always possible to satisfy an arbitrary sequence of READW and WRITEW commands without moving the head between commands. For instance, it is generally not possible to WRITEW more than once at the same node without moving the head. Thus we must establish for a particular template a set of rules limiting the possible sequences of reads and writes. We call such a set of rules an access sequence discipline.

4.2.9 Template Refinement Commands

The data access graph of an already existing file is assumed to be fixed. Because the data access technique cannot in general obtain complete information about the structure of the file, it must express what information it does know as a template rather than as a single graph.

When a new file is being created, however, the data access graph does not yet exist. One might say that it is implicit in the inner structure of the algorithmic part of the routine, but it is certainly not accessible to the data access technique, nor is it reflected in any pattern of data on a physical volume. Thus in this case the template represents real indeterminacy in the structure of the file, rather than simple lack of information.

The data access technique must, however, know enough about the file structure to be able to insert data words and control information at the proper points on the physical volume. For instance, it must know the length of a record it is about to write if it intends to place a record length indicator in the record header. Thus there must be some mechanism whereby the algorithm may inform the data access technique of any refinements it makes to the template name. To this end, we provide the command

REFINE TEMPLATE <PLIST>

If the current template name is <TNAME>, the new template name will be (<TNAME>), <PLIST>.

5. TRANSFERABILITY OF PROGRAMS USING THE DATA ACCESS REPRESENTATION

5.1 Characterization of Programs

We will, as before, consider a user program as consisting of an algorithmic part, a program assembly part, and a data access part. The program assembly part is normally executed only at the beginning and ending of a computational process. It does not mix with the algorithmic part nor with the data access part. We may consider the program assembly part as forming a partition of the program's execution into subunits which we shall call routines. A routine consists of algorithmic and data access statements inextricably intertwined. To achieve transferability for the data access part, we must find ways of altering it in place, without having to understand or modify the algorithmic part.

We do not know in general how the user program's algorithm works, nor even what the individual algorithmic statements mean. We will assume, however, that the program documentation is sufficiently detailed to allow us to construct a flowchart of the routine.

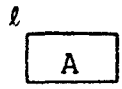
5.1.1 Pattern of Access Graph

We call such a flowchart a pattern of access graph. It can contain the following types of flowblocks:

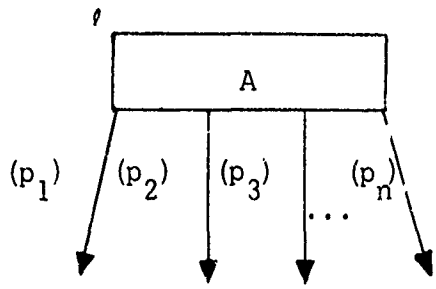
5.1.1.1 Algorithmic Command Flowblock

This represents a sequence of purely algorithmic statements where only the last statement may be a branch point and where only the first statement may have a statement label. The branch instruction, if it exists, must indicate explicitly the set of possible branch locations. This set must be finite and any element of the set must either be the next instruction after the

branch or be a statement label. As we do not presume to know how the algorithm works, we will consider all locations in the set as possible next instructions whenever control reaches the branch instruction. An algorithmic command flowblock is denoted by



If the first instruction has a statement label ℓ, it is written outside the upper left hand corner. If the last instruction is a branch instruction with n branch locations we draw n arcs exiting from the flowblock and label each with the probability that it will be chosen:



where $\sum_{i=1}^{n} p_i = 1$

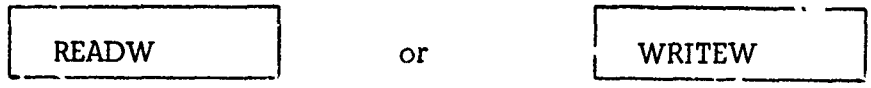
5.1.1.2 Head Movement Command Flowblock

There will be exactly one of these flowblocks for each MOVEH instruction in the routine. We denote it by a box with the instruction written inside:

where α is an arc label in the volume access graph.

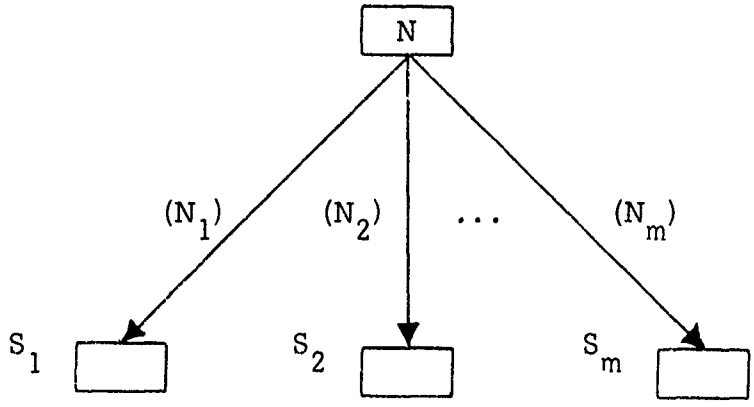
5.1.1.3 Data Access Command Flowblock

There will be exactly one of these flowblocks for each READW or WRITEW instruction in the routine. It is denoted by a box with the instruction inside:



5.1.1.4 Node-type Branch Flowblock

There will be exactly one of these flowblocks for each ON NODE instruction in the routine. ON NODE (N1, N2, ..., Nm) GOTO (S1, S2, ..., Sm) will be represented by



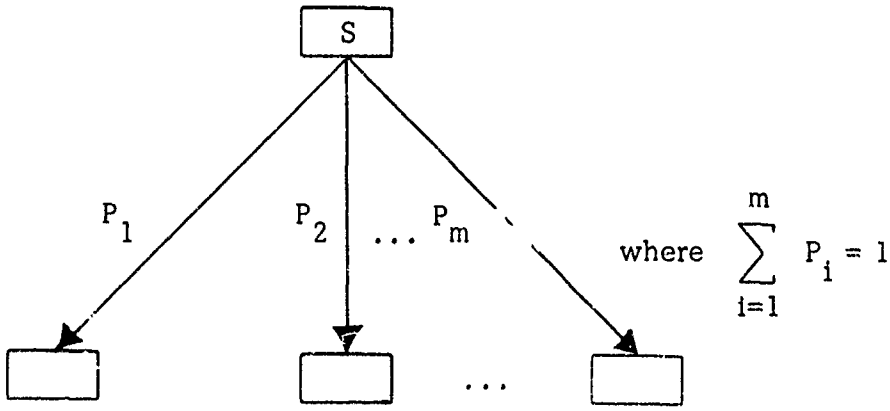
If two or more of the S_i are identical they may be represented by one arc labelled by a list of the node types.

5.1.1.5 Template Refinement Flowblock

There will be exactly one of these flowblocks for each REFINE TEMPLATE command in the routine:

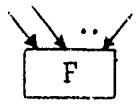
5.1.1.6 Start Flowblock

There will be one such flowblock. From it will be drawn arcs to all flowblocks where control could enter this routine. If there is more than one arc, probabilities will be assigned:



5.1.1.7 Finish Flowblock

There will be one such flowblock. All flowblocks representing instructions where control will leave the routine will have arcs going to it:



5.1.2 Context in a Pattern of Access Graph

One useful item which may be calculated for every flowblock in a pattern of access graph is the set of possible combinations of template name and node type that could be true of the access technique state when control enters the block of code corresponding to that flowblock. We call such a set the context of a data access flowblock.

Each element of a flowblock context will be of the form (t,n), where t is a template name and n is a node type.

We may calculate the context for every flowblock in the pattern of access graph by a fairly simple algorithm. Assume for simplicity that only one template name is allowed in the routine. This is the normal case, and routines which allow more than one template can be handled by a relatively straightforward extension of the algorithm to be given.

First we need, for the template accepted by the routine, a function

$$\sigma: A \times T \rightarrow 2^T$$

where A is the set of possible arc labels and T is the set of possible node types. For any $a \in A$ and any $t \in T$ the set $\sigma(a,t)$ contains the types of all nodes that can be reached from some node of type t by executing the command MOVEH a.

Suppose there are n flowblocks in the pattern of access graph, that the first flowblock is the start block, and that the nth flowblock is the finish block. Let $\Gamma: N \rightarrow 2^N$, where $N = \{1, 2, \ldots, n\}$, describe the flow paths in the pattern of access graph. Let $C_1, C_2, \ldots, C_n$ be sets of

node types. Initialize $C_2, C_3, \ldots, C_n$ to be null sets and C_1 to contain the types of the possible initial states for the data access technique when the routine is entered.

Define a path p through the pattern of access graph to be a sequence of flowblock numbers $p = (f_1, f_2, \ldots, f_q)$ where for every i such that $1 \leq i \leq (q-1)$,

$$f_{i+1} \in \Gamma(f_i)$$

Call f_1 the initial block and f_q the terminal block of the path. Associate with any such path a set of node types T_p which we shall call the path context.

For any given path $p = (f_1, f_2, \ldots, f_q)$ with context T_p, define an extension of the path by the following nondeterministic algorithm:

1. If $\Gamma(f_q)$ is null, there is an error. Otherwise select an f_{q+1} from $\Gamma(f_q)$.

2. If f_q is anything but MOVEH or ON NODE go to step 5.

3. If f_q is MOVEH a, for some a, then set

$$T_p \leftarrow \bigcup_{t \in T_p} \sigma(a,t)$$

and go to 5.

4. If f_q is an ON NODE instruction and the arc (f_q, f_{q+1}) has label t then set $T_p \leftarrow t$.

5. Set $z \leftarrow T_p$.

6. Set $T_p \leftarrow T_p - C_{q+1}$. Set $C_{q+1} \leftarrow C_{q+1} \cup z$.

We can now use this process of path extension to calculate the context of each flowblock:

1. Start with a single path, of length 0, namely the path (1) consisting of the start block. Let T_p for this path be C_1. Let B be a set containing this path as its only member.

2. Extend every path in B. Let C be the set of all paths and associated path contexts that can be formed in this way.

3. Subtract from C all paths with null contexts.

4. If C is null, stop.

5. Set $B \leftarrow C$.

6. Go to 2.

It can be proved in a straightforward but tedious manner that this process always stops in a finite number of steps and that the results are independent of the order in which paths are extended in step 2. The resulting C_1, C_2, ..., C_q are the flowblock contexts we desire.

5.2 Characterization of Program Execution

Once we have the pattern of access graph for a routine we may analyze all of the ways that routine could possibly access secondary storage. Any path through the pattern of access graph which starts with the S block and ends with the F block represents a possible flow of control through the routine. We call any such path a control history, and represent it as $(h_1, h_2, \ldots, h_m)$ where each h_i is a flowblock number. Of course, $h_1 = 1$ and $h_m = q$, where q is the number of flowblocks.

5.2.1 Information Flow During Program Execution

As a first step toward separating the data access part from the algorithmic part of a control history, let us examine how each flowblock affects the flow of information between the algorithm and the data access technique. We may divide the flowblocks into the following categories:

5.2.1.1 Type A Flowblocks

Algorithmic flowblocks have no contact with the data access technique. Thus the internal workings of the data access technique, the current state of the data access graph, and the current template have no effect on the correct execution of an algorithmic flowblock. Conversely, the data access technique cannot be directly affected by the execution of an algorithmic flowblock.

5.2.1.2 Type A → D Flowblocks

These represent one way information flow from the algorithm to the data access technique. Included in this type are MOVEH and REFINE TEMPLATE. These commands set no registers nor alter any data directly accessible to the algorithm, and thus do not directly affect the execution of the algorithm.

5.2.1.3 Type D → A Flowblocks

These represent one-way information flow from the data access technique to the algorithm. This type contains the ON NODE command.

5.2.1.4 Type A & D Flowblocks

These represent two-way information flow between the data access technique and the algorithm. Included in this type are READW and WRITEW. In both cases data must flow between the storage volume and some register accessible to both the algorithm and the data access technique.

5.2.2 Data Access History

Let us now examine, for any control history, that subsequence composed of all flowblocks of the following types:

A → D

MOVEH
REFINE TEMPLATE

D → A

ON NODE

A & D

READW
WRITEW

We call any such subsequence a data access history. It represents those operations that require the attention of the data access technique during the course of program execution.

We stated at the outset that programs which depend on time intervals rather than time sequences are not transferable. We shall now make use of that restriction to decouple somewhat the actions of the algorithm and the data access technique. We shall add a FIFO queue, the Data Information Access Queue (DIAQ). When the user program issues a command of type A → D it will be placed on the DIAQ and the user program will then continue as if the command had been processed by the DAT. The DAT will, at its leisure, remove commands from the DIAQ and execute them.

As long as the user program issues commands only of type A or A → D the algorithm and the data access technique may proceed independently of each other. The DAT may allow commands to pile up in the DIAQ and thus the current node and current template may not be what the algorithm expects. Commands of type A or A → D , however, do not transmit information from the DAT to the algorithm and thus the algorithm cannot be affected by the discrepancy. For clarity, let us call the node and template that should be current the current node and current template and the node and template that are actually current the lagging node and lagging template.

A user program command of type A & D requires that the DAT virtual head be at the current node. The program must wait while the DIAQ is emptied. The DIAQ contains a string of MOVEH and REFINE TEMPLATE commands. The result of executing, in order, all commands in the string will be that the virtual head will have been moved from the lagging node to the current node and the lagging template will have been refined to the current template. All details of the path taken in getting from the lagging node to the current node,

however, will have been lost. Thus any other string of A → D commands which has the net effect of moving the lagging node and template to the current node and template will produce precisely the same result. We shall use this freedom later.

In summary, when a command of type A & D is issued by the user program that program must halt while the lagging node and template are advanced to the current node and template. This process is defined by the contents of the DIAQ. The DAT may accomplish this by executing all commands in the DIAQ, or by executing an equivalent string of commands.

The only D → A command is ON NODE. This requires that the virtual head be moved to the current node in order to test its type. As with A & D commands, any path to the current node will do.

A data access history may thus be characterized as

$$(n_0, T_0; P_1 N_1 P_2 \cdots N_{m-1} P_m N_m)$$

where

n_0 is the starting node

N_i $1 \leq i \leq m$ are node accesses

P_i $1 \leq i \leq m$ are strings of A → D commands

T_0 is the initial template

Node n_0 is the current node and T_0 the current template as initialized by the control section of the user program. T_0 is a macro and n_0 one of its external nodes.

Node accesses are in general any A & D or D → A commands. We must make an exception, however, for ON NODE. ON NODE causes a branch to one of a set of locations depending on the current node type. The result of this branch is, however, implicit in the control history from which the data access history was abstracted. Thus we must replace ON NODE (T_1, T_2, T_3) GOTO (S_1, S_2, S_2) with TYPE = $\{T_1\}$ if the next block in the control history was S_1 and with TYPE = $\{T_2, T_3\}$ if the next block was S_2. Thus the only legal node accesses are READW, WRITEW, and TYPE = $\{T_1, T_2, \ldots\}$.

Each P_i can be thought of as an operator on the set of template names:

$$P_i : t \rightarrow t'$$

where

t is a template name or ERROR

t' = t if P_i contains no REFINE TEMPLATE commands

t' = ERROR if a REFINE TEMPLATE command is incompatible with t

t' = the template name produced by applying in order all REFINE TEMPLATE commands in P_i to t

Let $T_f = P_m P_{m-1} \ldots P_1 T_0$ be the final template for this data access history. If T_f = ERROR then the data access technique will have aborted the job. Let us now consider a graph $g \in T_f$. g has a definite number of nodes, say q of them, which are labelled with the integers 1, 2, ..., q. Construct a sequence of (m + 1) node indexes as follows:

1. set $g_0 \leftarrow n_0$; set $i \leftarrow 0$

2. if $i \geq m$, stop

3. set $g_{i+1} \leftarrow P_{i+1} g_i$

4. set $i \leftarrow i+1$; go to 2 .

where $P_i g_i$ is the node reached by taking the path defined by P_i from node g_i .

If each g_i is paired with the corresponding N_i the pair $a_i = (g_i, N_i)$ defines a transfer of information between the algorithm and a particular word on a storage volume. The sequence $a = (a_0, a_1, \ldots, a_m)$, called a word access history, expresses the total interaction between the algorithm and the data on secondary storage.

Thus for every data access history there is a set of word access histories, one for each graph in the final template for that data access history.

5.3 Associate Graph Transformations

Inasmuch as the data access representation can be programmed for a variety of machines and operating systems it is more transferable than, say, machine code. The same can be said, however, for most other data management systems which interpose some software between the user program and the physical I-O devices. We want more than just a clean language. We wish to be able to move a data file to a different device, alter a program using this file to take advantage of the characteristics of the new device, and yet not alter nor be required to understand the algorithmic part of the program.

As was shown in the previous section, the algorithm's use of the data file consists of a sequence of accesses to specific nodes of the specific graph representing the data file. The only permissable types of access are a test of node type and a read or write of the node's data content. The algorithm is unaffected by the way in which the virtual head is moved to each successive node.

If the new device is significantly different from the old device, then one would expect that the "same" data file would be represented by different graphs on the two devices. The nodes mentioned in any word access history for the unmodified program would have exact counterparts in the new graph. The new graph might have extra nodes, and it will almost certainly have a different arrangement of arcs and arc labels.

We shall for the moment ignore the possibility of extra nodes and concentrate on the problem of graphs with the same nodes but a different arrangement of arcs.

5.3.1 Associate Graphs

It is first necessary to define precisely what is meant by two graphs with the same nodes but different arcs.

AG: Definition

Let G and G' be data access graphs. G' is an associate of G if and only if

AG 1: $\eta(G) = \eta(G')$

AG 2: $\forall n \in \eta(G)$,

$$\text{type } (n_G) = \text{type } (n_{G'})$$

$$\text{access restriction } (n_G) = \text{access restriction } (n_{G'})$$

$$\text{value } (n_G) = \text{value } (n_{G'})$$

where for any data access graph Q, $\eta(Q)$ is a set of integers uniquely labelling each node in Q and where n_Q is the node in Q with label n.

Suppose G and G' are associates. G and G' can be thought of as a single graph with two colored sets of arcs. The red arcs belong to G and the green to G'.

Suppose that a program accesses the data file represented by G and that one of its data access histories is

$$(g_0, T_0; P_1 N_1 P_2 N_2 \ldots P_m N_m)$$

G, of course, belongs to $T_F = P_m P_{m-1} \ldots P_1 T_0$. This data access history, in conjunction with G, gives us the data we need to construct a word access history:

$$((g_0, -), (g_1, N_1), \ldots, (g_m, N_m))$$

where for every $0 \leq i \leq m-1$

$$P_{i+1} g_i = g_{i+1}$$

Suppose now that this program is to be altered to access G' in place of G. We will examine the same data access history as before and suppose that the program has not been altered. In order that the algorithm operate correctly, the word access history

$$((g_0', -), (g_1', N_1), \ldots, (g_m', N_m))$$

must be such that

$$g_i' = g_i \quad \text{for} \quad 0 \leq i \leq m$$

The path functions P_i, of course, do not apply to the new graph G'. We need a new set of path functions P_i' such that

$$g_{i+1}' = p_i' g_i \quad \text{for} \quad 0 \leq i \leq m-1$$

Remember now that P_i is not a path in G but a path function consisting of a number of MOVEH commands. Its action differs for different nodes. P_i may be described as follows

$$P_i : \{1, 2, \ldots, q\} \rightarrow \{\underline{\text{ERROR}}, 1, 2, \ldots, q\}$$

where $P_i(b) = \underline{\text{ERROR}}$ if, starting from node b, the MOVEH commands in P_i would cause an error.

Since path functions are right associative and since each P_i is composed of a string of single arc functions, we need a function γ such that

$$\gamma : (a, n) \rightarrow F_a'$$

where

a is an arc label

n is a node index

and where F_a' is a path function for G' such that

$$F_a'(n) = a(n)$$

In our two-colored graph analogy, suppose we are at node n and there is a red arc exiting from n and labelled a. At the other end of the arc is another node, say t. There is also (strong connectivity) a path from n to t using only green arcs. $\gamma(a,n)$ will be a string consisting of the labels on these green arcs.

Thus if every MOVEH a issued by the user program for data access graph G is replaced by MOVEH $\gamma(a,n)$, where n is the current node at the time the command is issued, the program will run successfully using G'.

Unfortunately, neither the user program nor the data access technique knows the unique name of the current node. At most they can know the type of the current node. Thus if it is possible to transfer from G to G', γ must obey the following restriction

$\gamma(a,n_1) = \gamma(a,n_2)$ whenever n_1 and n_2 have the same type for every arc label a in G. If γ satisfies this we say that the transformation from G to G' is <u>context consistent</u>. A context consistent transformation will be denoted by $\delta(a,t)$, where t is a node type and $\delta(a,t) = \gamma(a,n)$ for some node n having type t.

Let us formalize this.

CCGT: Definition

Suppose the two graphs G and G' are associates and are related by the transform

$$\delta : PCS \times NTS \rightarrow \{ PCS'^* \cup \langle NIL \rangle \}$$

where PCS is the set of arc labels in G, NTS is the set of node types in G, and PCS* is the set of all finite strings of arc labels from G'. Let elements of PCS and PCS' be viewed as right associative operators on the sets of nodes of G and G', respectively. ϒ is a context consistent graph transform if and only if

CCGT 1: $\forall a \in PCS, \forall t \in \{ Z \mid \delta(a,Z) = \langle NIL \rangle \}$,
$\forall n \in \{ x \mid x \in \eta(G) \text{ and type}(x) = t \}$,
$a\ n = \delta(a,t)\ n$

In other words, whenever there is an arc with label a leaving a node of type t in G there is an equivalent path in G' denoted by the string of arc labels $\delta(a,t)$.

5.3.2 Associate Templates

In the previous section we dealt with single graphs. Any practical transformation, however, must apply to all graphs in a template. This requires the following generalization:

AT: Definition

Let (T, TNC) stand for a template with name T and naming convention TNC. Let (T', TNC') be another template. Let

AT 1: $|T| = |T'|$

and $|TNC| = |TNC'|$

T' is an associate of T if and only if there exist 1-1 onto functions

$$\omega: T \to T'$$
$$\sigma: TNC \to TNC'$$

such that

AT 2: $\forall g \in T$, $\omega(g)$ is an associate of g

AT 3: $\forall N \in TNC$, $\omega: N \to \sigma(N)$

CCT: Definition

T' is context consistent with T if and only if

CCT 1: T' is an associate of T

CCT 2: There exists a function δ which for every $g \in T$ is a context consistent graph transform to $\omega(g)$.

5.3.3 Correct Localized Transfers

We are now ready to describe a process which will alter a user program to compensate for a change to an associate template. This process is quite mechanical, requiring no knowledge of the program's function or logic. The results are guaranteed correct. Furthermore, if the process is coupled with knowledge of the possible paths of control through the program, their significance, and the probability each will occur then a solution may be obtained which is nearly optimal among solutions which do not alter the algorithm of the original program.

First, let us define a formal framework in which to discuss the process. Let UP be a user program containing q statements. Let P be the pattern of access graph for UP. When a flowblock in P corresponds to the r-th command in UP then let that flowblock be labelled as flowblock r. Any legal control path through UP must start at command 1 and end at command q.

For simplicity, we will not condense blocks of non-branching type A commands into single flowblocks. As a result there will be a 1-1 correspondence between commands in UP and flowblocks in P.

Suppose UP accesses a file F. We know the template name, T, of F. We know the template naming convention, TNC, of F. This is the set of T and all refinements of T. Furthermore, for every graph $q \in T$, we know the starting node, that is, there is a function S such that

$$\forall g \in T, \quad S(g) \in \eta(g)$$

where $\eta(g)$ is the set of all nodes of g. Now T, TNC, and S characterize the file F, and we call the triple (T, TNC, S) a file. Similarly we call the quadruple (P, T, TNC, S) a file use.

As has been described before, given a pattern of access graph P we may generate the set of all paths through P which start at block 1 and end at block q. Each path is described by listing, in order, the indices of all blocks through which it passes. We call such a path description a control history. From a control history we may produce a data access history by converting, in order, all indices through use of the following rules:

1. If the index refers to a type A command, discard the index.

2. If the index refers to a type A & D command, replace it with the command.

3. If the index refers to a MOVEH or REFINE TEMPLATE command, replace it with the command.

4. If the index refers to an ON NODE command, check the next index. Replace the current index with the statement TYPE = C, where C is the set of all node types that could have caused control to branch to the next index.

We describe the entire process of going from a pattern of access graph P to a set of data access histories DAH by the function χ.

$$\chi(P) = DAH$$

where the domain of χ is the set of all pattern of access graphs.

The virtual head, using an element of DAH as input, operates on the file described by the triple (T, TNC, S) . We embody the rules of the data access representation in two functions ξ_1 and ξ_2 . ξ_1 is characterized as follows

$$\text{given} \quad F = (T, TNC, S) , \quad \text{given} \quad DAH ,$$
$$\forall h \in DAH , \quad \xi_1 (F,h) = T_f \in TNC$$

that is, ξ_1 extracts from a particular data access history all of the REFINE TEMPLATE commands and produces the refined template they imply. ξ_2 describes the movement of the virtual head; it transforms a particular data access history into a set of word access histories, one for each graph in T_f :

$$\text{given} \quad F = (T, TNC, S) , \quad \text{given} \quad DAH ,$$
$$\forall h \in DAH , \quad WAH (h) = \{ \xi_2 (h,g) \mid g \in \xi_1 (F,h) \}$$

The range of ξ_2 is the set of legal word access histories and the special element <ERROR> , which implies that the commands in h were incompatible with either the arc labels or the node access restrictions.

Normally, if UP is well written, one would not expect a WAH to contain < ERROR > as it implies that there is a path through UP which would cause an I-O error. This situation could arise, however, because P represents a simplification of the true control structure of UP. If, for some history h , WAH (h) contains < ERROR > then h is logically inconsistent with the data file structure. We call a data history, h, erroneous if WAH (h) contains < ERROR > . If it can be shown somehow that h will never be taken by UP then it is spurious.

Let us now describe a process of altering a user program, UP, which accesses a file, F, so that the altered program, UP', correctly accesses a file F' , which is an associate of F . Let F = (T, TNC, S) and let F = (T', TNC', S') . The fact that F' is an associate of F implies the existence of ω, σ, and δ such that

$$\omega : T \rightarrow T'$$
$$\sigma : TNC \rightarrow TNC'$$
$$\delta : PCS \times NTS \rightarrow \{ PCS' \cup <NIL> \}$$

We shall further require that $\forall g \in T$, $S(g) = S'(\omega(g))$

Produce UP' by copying UP and then making the following changes to the copy:

1. If a REFINE TEMPLATE command is found, it will be in the form REFINE TEMPLATE φ, where φ: TNC → TNC . Replace φ by φ' . where $\forall N \in TNC$, $\sigma(\varphi(N)) = \varphi'(\sigma(N))$.

2. If a MOVEH a command is found, for any arc label a, replace it by

   MOVEH $\left[\delta(a,t_1)(t_1) , \delta(a,t_2)(t_2) , \ldots , \delta(a,t_n)(t_n)\right]$

   where $\{ t_1, t_2, \ldots, t_n \}$ is the context of the corresponding flowblock in P.

3. Do not alter any other instructions.

We have taken two notational liberties in the above. First, REFINE TEMPLATE usually has an argument that looks like (PARM = 5) rather than some function over the set of template refinements, TNC. It should be fairly clear, however, that these are simply two forms of the same thing. One form is more convenient for programming, one is more convenient for theoretical work.

Secondly

$$\text{MOVEH } \left[a_1(t_{11},t_{12},\ldots),\ a_2(t_{21},t_{22},\ldots),\ \ldots,\ a_n(t_{n1},t_{n2},\ldots)\right]$$

is equivalent in action to

```
          ON NODE (t11,t12,...,ELSE)  GOTO (NEXT, ..., NEXT, ¢ON1)
          MOVEH    a1
          GOTO     ¢OUT

¢ON1      ON NODE (t21,t22,...,ELSE)  GOTO (NEXT, ..., NEXT, ¢ON2)
          MOVEH    a2
          GOTO     ¢OUT

¢ON2            .
                .
                .

¢ONn-1    ON NODE (tn1,tn2, ...) GOTO (NEXT)
          MOVEH    an

¢OUT      NOOP
```

That is, it will cause the virtual head to move along arc a_1 if the current node is of type $t_{11}, t_{12}, \ldots,$ along arc a_2 if of type $t_{21}, t_{22}, \ldots,$ and so forth. If the type of the current node is not listed, an error occurs. This form of MOVEH is introduced in order to retain a one to one correspondence between control histories in P and P'.

While we won't rigorously prove it, it is fairly easy to see that UP' will perform the same calculations and get the same answers as UP. First, UP and UP' have exactly the same control structures. The only alterations were one for one replacements of REFINE TEMPLATE and MOVEH commands. Because neither of these commands cause branching nor do their replacements, there is no change to the number of flowblocks in the pattern of access graph nor to the arcs between these flowblocks. Thus, if equivalent flowblocks in P and P' are given the same integer labels, the control histories for UP and UP' will be identical. As a result, there will be an obvious 1-1 correspondence τ between DAH and DAH'

$$\tau: \text{DAH} \rightarrow \text{DAH}'$$

where for any $h \in \text{DAH}$, h and $\tau(h)$ result from the same control history, just with different instructions in some of the flowblocks.

Now we must show that equivalent control histories in P and P' compute the same thing. First, no type A instruction was altered. Secondly, MOVEH and REFINE TEMPLATE are $A \rightarrow D$ instructions. Short of causing an error, they convey no information to the program, and hence cannot directly affect the result of any computation. The only real problem is with the A & D commands READW, WRITEW, and ON NODE. These commands are the same in UP and UP', but they act upon different nodes because the files are different.

All this can be summed up in the following way: For any $h \in DAH$, h and $\tau(h)$ must correspond to equivalent computations.

h has the general form

$$h = (P_1 N_1 F_2 N_2 \ldots P_m N_m)$$

where for $1 \leq i \leq m$

1. N_i = READW, WRITEW, or ON NODE

2. P_i = null op

   or

   $p_{i1} p_{i2} \cdots p_{ik_i}$

   where for $1 \leq j \leq k_i$,

   p_{ij} = MOVEH a_{ij}
   or
   REFINE TEMPLATE φ_{ij}

$\tau(h)$ has the general form $\tau(h) = P'_1 N_1 P'_2 N_2 \ldots P'_m N_m$

where for $1 \leq i \leq m$

P'_i = null op if P_i = null op

otherwise $P'_i = p'_{i1} p'_{i2} \cdots p_{ik_i}$

where for $1 \leq j \leq k_i$

p'_{ij} = REFINE TEMPLATE φ' if p_{ij} = REFINE TEMPLATE φ

$$p'_{ij} = \text{MOVEH}\ \left[\delta(a_{ij}, t_1)\ (t_1),\ \delta(a_{ij}, t_2)\ (t_2),\ \ldots,\ \delta(a_{ij}, t_z)\ (t_z)\right] \quad ,$$

where $\{t_1, t_2, \ldots, t_z\}$ is the command context, if $p_{ij} = \text{MOVEH}\ a_{ij}$

The data access technique, ξ_2, operating on h produces, for each graph g in $\xi_1(F,h)$, a word access history

$$\xi_2(h,g) = ((n_0, -), (n_1, N_1), \ldots, (n_m, N_m))$$

where

1. $n_0, n_1, \ldots, n_m$ are node indices in g,
2. for $1 \leq i \leq m$, $n_i = p_i\ n_{i-1}$
3. $n_0 = S(g)$

Similarly, for any g' in $\xi_1(F', \tau(h))$,

$$\xi_2(\tau(h), g') = ((n'_0,), (n'_1, N_1), \ldots, (n'_m, N_m))$$

where

1. $n'_0, n'_1, \ldots, n_m$ are node indices in g'.
2. for $1 \leq i \leq m$, $n'_i = P'_i\ n'_{i-1}$

Now we know, since F' is an associate of F, that if UP processes graph $g \in \tau$, under the same conditions UP' will process $\omega(g)$.

We thus want

$$\xi_2(h,g) = ((n_0,-), (n_1, N_1), \ldots, (n_m, N_m))$$

and

$$\xi_2(\tau(h), \omega(g)) = ((n'_0, -), (n'_1, N_1), \ldots, (n'_m, N_m))$$

to be equivalent operations. Since (by condition AG2) nodes of the same index in two associate graphs have the same type, value, and access restriction these two word access histories will be equivalent if for every $0 \leq i \leq m$, $n'_i = n_i$.

Fairly obviously, the following two conditions would guarantee this by induction

IC1: $n'_0 = n_0$

IC2: for $1 \leq i \leq m$, if $n_{i-1} = n'_{i-1}$, then $n_i = n'_i$

IC1 is, of course, guaranteed true because we required that associate graphs have the same starting node indices, i.e.,

$$\forall g \in T, S(g) = S'(\omega(g))$$

To guarantee IC2 true we must prove that for any i such that $1 \leq i \leq m$,

$$P_i n_{i-1} = P'_i n_{i-1}$$

But this is true if for every j such that $1 \leq j \leq k_i$,

$$p_{ij}\, n_{i-1} = p'_{ij}\, n_{i-1}$$

If p_{ij} = REFINE TEMPLATE so is p'_{ij} and $p_{ij}\, n_{i-1} = p'_{ij}\, n_{i-1}$

If, however, p_{ij} = MOVEH then

$$p'_{ij}\, n_{i-1} = p_{ij}\, n_{i-1}$$

is true by condition CCGT1 in the definition of the context consistent graph transform δ.

Thus, since IC1 and IC2 are true, it must be true that

$$\forall h \in DAH, \forall g \in \xi_1(F,h), \xi_2(h,g) = \xi_2(\tau(h), \omega(g))$$

Thus we know now that the type A & D commands in UP and UP' produce equivalent results. Since the type A commands in UP and UP' are identical, the type A → D commands don't affect the program, and the A & D commands in UP and UP' produce equivalent results, UP and UP' must produce equivalent results.

It should be pointed out that while the existence of <u>one</u> function $\delta(a,k)$ is assured, in general there will be <u>many</u> alternate paths that consistently replace an arc labelled a emanating from all nodes of type t. The programmer transforming UP is, of course, free to select the most efficient context consistent path.

5.4 Summary

In Chapter 5 we characterized the interactions between the algorithmic and data access portions of a user program. We then defined a particularly simple data file transformation where the new file has the same nodes as the old but a new set of head movement primitives. We showed that a certain consistency had to exist between arc labels in a template and in its transform in order for a simple transformation rule to exist. We called this context consistency. A template which was obtainable from another template by a context consistent transform was called an associate template. We then gave a mechanical procedure whereby a user program could be altered to correctly access a transformed file and gave an informal proof of that correctness.

Such a mechanical procedure for altering programs is precisely what we set out originally to produce. As such, it represents a significant advance in the field of transferable data management facilities. It is by no means a complete solution, however, Associate transformations are fairly restricted in scope, and we suspect that most practical problems in transferability will require more general transforms. We must, therefore, find more transforms and then define mechanical procedures for altering programs to compensate for them. It should be clear from the straightforward way in which the associate template program transformation was derived and proven that we have hardly begun to stretch the limits of our theoretical framework.

We are especially interested in the following:

1. Files often contain blocks of control information. In the data access technique this would be represented by a string of control nodes, say $(n_1, n_2, \ldots, n_m)$. The user program

might access all these nodes and then extract certain information from all of them. Mathematically, one could say that the program calculates the functions $d_1 (n_1, n_2, \ldots, n_m)$, $d_2 (n_1, n_2, \ldots, n_m)$, ...

In the transformed file, however, the old string of control nodes is replaced by a new string of nodes $(n'_1, n'_2, \ldots, n'_p)$. No individual node n'_i has any obvious relationship to any node in the old string, but the new string contains "the same information", i.e., it is possible to define functions $f_1, f_2, \ldots$ such that $f_i (n'_1, n'_2, \ldots, n'_p) = d_i (n_1, n_2, \ldots, n_m)$.

2. Sometimes control information is shifted some limited distance within a file. For instance, IBM OS/360 has a header defining the length of a variable length record and CDC SCOPE has a trailer. In both cases the size of the record is bounded. In going from an IBM to a CDC system, one could pick up IBM's header information by spacing to the end of a CDC record, reading the trailer, then spacing back to the beginning of the record. This is clearly not desirable from a performance standpoint. There appears to be a variety of interesting alternatives which should be investigated.

3. Suppose the transformed file has a different word width. There will then no longer be a 1-1 relationship between equivalent nodes. If the word width in one file is an integral multiple of the width in the other then a relatively simple transformation is possible. Consider, however, the case of

going from 3-bit octal to 4-bit hexadecimal, where 4 octal nodes correspond to 3 hex nodes This will require a more sophisticated transform.

4. One problem with context consistency is that in order to check for it one must not only know the original file but also its transformed version. We would like to develop criteria which will guarantee that a given file has an interesting and non-trivial set of context consistent associates without having to derive any of these associates.

6. ADDITIONAL MACHINERY FOR THE DATA ACCESS REPRESENTATION

The following is a collection of special techniques and abbreviations that were developed in the process of writing the next two chapters. These are largely items of convenience which allow some of the complexities of real-world systems to be expressed compactly; they do not really add anything new to the theoretical framework established in Chapter 5. XFORM, introduced in section 6.1 of this chapter, is an exception. It allows us to model the dynamic changes in data file structure that occur when a data file is written into or edited. We show that this does not affect the results of Chapter 5 provided a "commutativity" relation holds between the possible transformations caused by XFORM and any transforms to associate templates required for transferability.

6.1 Modification of Data Structures

Heretofore we have always assumed that between OPEN and CLOSE there was only one graph which represented the data file. We dealt with templates only because it was assumed that the user program and/or data management routine did not know which particular graph applied. This assumption works well when the data file is being read or when it is being modified in place. It shows signs of strain when a new file is being created. It breaks down completely, however, for IBM partitioned data sets. A partitioned data set contains a directory of names, each of which points to a sequence of data records called a member. It is possible to delete members, create new members, and replace members with new ones of a different size. It is not reasonable to describe this by a single data access graph.

We would like, therefore, to consider a class of data access graph transformations. This class must be rich enough to cover the specific transformations allowed in the access methods we are studying, yet restricted sufficiently that they don't invalidate the analytical tools we have developed.

It should first be noted that useful transformations do not alter the basic structure of a data file That is, replacing a member of a partitioned data set still leaves it a partitioned data set. Thus we need only consider transformations which map the members of a template into other members of the same template.

The new command allowed in a user program will be

XFORM θ

where θ is a transform .

First, some definitions.

D1. For any data access graph g , $\eta(g)$ is the set of all nodes in g .

D2. For any data access graph g and any set of node types T , $\eta(g,T) = \{n \mid n$ is a node of g and type $(n) \in T\}$.

D3. Transformations generally depend on the current node. For instance, in order to delete a member of a partitioned data set it is necessary for the current node to be in the directory entry for that member. Thus we will consider a transformation θ to be valid only if the current node type is one or a set of types C , called the context of θ.

D4. A particular transformation θ is valid only for graphs of a certain structure. For any transformation θ there is a template T_θ which <u>covers</u> θ.

D5. A transformation θ with context C_θ and covering template T_θ has the form

$$\theta = (\theta_1, \theta_2, \theta_3)$$

where θ_1, θ_2, and θ_3 are functions.

D5.1 $\forall g \in T_\theta, \forall n \in \eta(g, C_\theta)$

1. $\theta_1 (g,n) \subseteq \eta(g)$
2. $n \in \theta_1 (g,n)$

That is, for a graph g and a particular current state n, θ_1 selects a subset of the nodes in g. These will be called the <u>passive nodes</u> and will be unaffected by the transformation θ. The current state must be one of those passive nodes.

D5.2 $\forall g \in T_\theta, \forall n \in \eta(g, C_\theta)$

$$\theta_2 (g,n) = g' \in T_\theta$$

That is, for any given graph in the covering template T_θ and for any legal current node, θ_2 selects a new graph, also in T_θ.

D5.3 $\forall g \in T_\theta$, $\forall n \in \eta(g, C_\theta)$, $\forall m \in \theta_1(g,n)$

1. $\theta_3(g,n,m) = m' \in \eta(\theta_2(g,n))$
2. type (m) = type (m')
3. value (m) = value (m')

That is, passive nodes are unaffected by the transformation. θ_3 identifies passive nodes in the old graph with identical nodes in the new graph.

Now that we have defined what a transformation is, let us see how it affects a user program that employs one. Allow XFORM θ to be represented by a type $A \rightarrow D$ flowblock in the pattern of access graph. In order for θ to work without error, C_θ must contain the context of the flowblock representing XFORM θ. This guarantees that the current state when the transform is to be made is one for which the transform is well defined.

As before, the pattern of access graph generates a set of data access histories, one for each possible flow of control through the program. Let us consider one such history:

$$(n_0 T_0 ; P_1 N_1 P_2 N_2 \ldots P_m N_m)$$

Suppose that P_i = XFORM θ and that no other P contains an XFORM.
Suppose that $T_\theta = P_{i-1} P_{i-2} \ldots P_1 T_0$.

Now for this data access history, for every $g \in T_\theta$ there is a word access history

$$((n_0,-)\ ,\ (n_1,\ N_1),\ \ldots,\ (n_m,\ N_m))$$

where

1. for $0 \leq j \leq i - 2$

   a. $n_j \in \eta(g)$

   b. $n_{j+1} = P_{j+1}\ n_j$

2. $Z = P_i\ n_{i-1}$

   $g' = \theta_2\ (g,\ Z)$

   $n_i = \theta_3\ (g,\ Z,\ Z)$

3. for $i \leq j \leq m - 1$

   a. $n_j \in \eta(g')$

   b. $n_{j+1} = P_{j+1}\ n_j$

So far we have done nothing more than pin down in precise notation exactly what XFORM does to a program. We must now show that a program transfer to an associate template is not damaged by XFORM.

First, a review of associate templates is in order. Remember that associate templates T and T' are related by a 1-1 onto function

$$\varphi : T \rightarrow T'$$

which establishes a correspondence between any graph $g \in T$ and some graph ${}_a g = \omega(g) \in T'$. We further assumed a correspondence between nodes of g and ${}_a g$ such that node i of g corresponded to node i of ${}_a g$. The only interesting template associations were those that were context consistent, that is, there was a function

$$\delta : P \times Q \rightarrow P^*$$

where

- P is the positional command set for T
- Q is the set of node types in T
- P^* is the set of all finite strings of positional commands for T'

This function relates arcs in g to equivalent paths in ${}_a g$.

We now make a very reasonable restriction and shall then show that it is sufficient to avoid damage:

R1. Suppose T and T' are associate templates, XFORM θ is covered by T and XFORM θ' is covered by T. θ and θ' are equivalent if

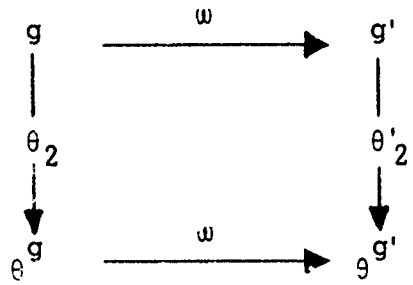
R1.1 $\theta_1 = \theta'_1$

R1.2 $\forall g \in T$, $\forall t \in C_g$, $\forall n \in \eta(g, \{t\})$,

$$\theta'_2(\omega(g), n) = \omega(\theta_2(g, n))$$

Sort of a "commutativity".

This is illustrated by the diagram



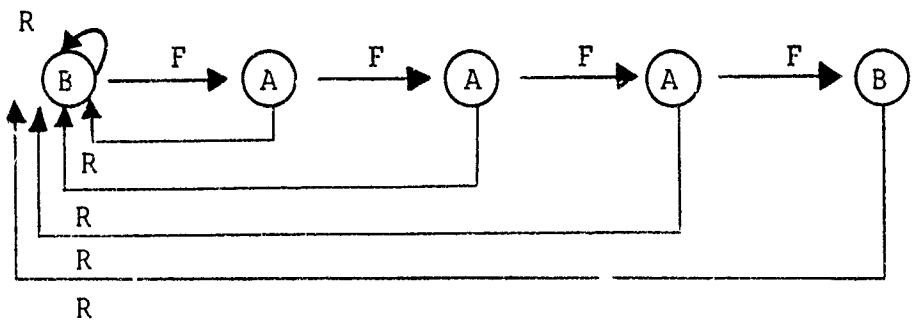
R1.3 $\theta_3 = \theta'_3$

Restriction R1 can be paraphrased that θ must transform associate graphs into associate graphs.

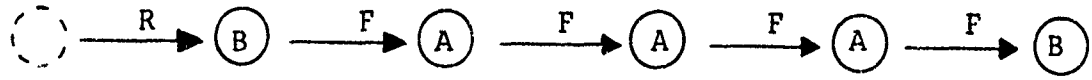
The result of R1 is that if equivalent data access histories for the old and the new user program start out accessing associate graphs they will at each step of the way continue to access associate graphs, which was all we needed.

6.2 Representative Nodes

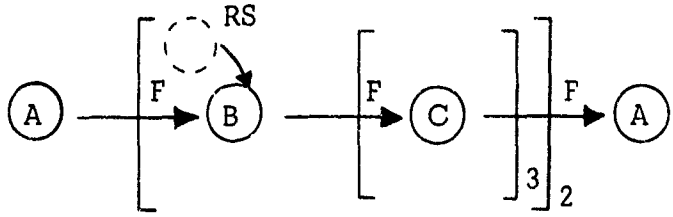
In a graph of the following structure



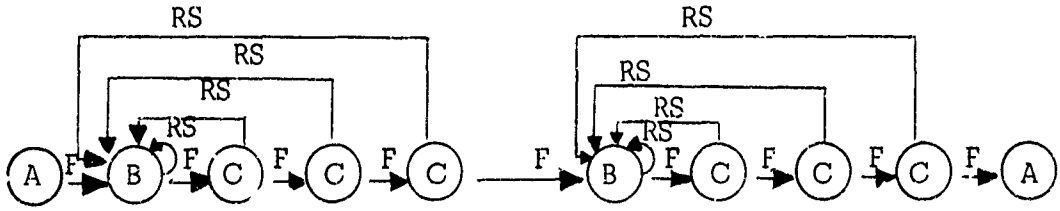
it would be convenient to be able to compactly represent the fact that all nodes have an R link to the first node. We will do this by introducing a representative node, which we draw as a dashed circle:



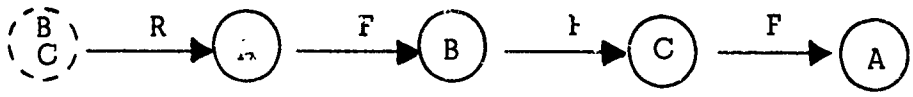
Representative nodes that do not appear within an iteration bracket apply to all nodes in the graph. Representative nodes that **appear within an** iteration bracket, however, apply only within that bracket. For instance



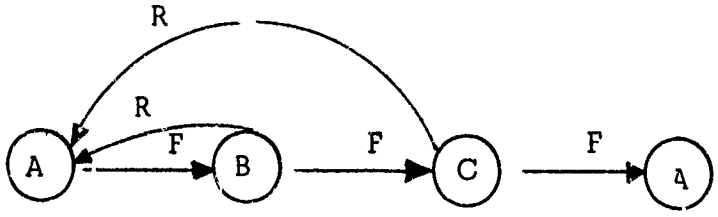
is equivalent to



If one or more node type designations appear within a representative node, that node represents nodes only of those types. For instance



is equivalent to



6.3 Choice Brackets

Suppose $P_1, P_2, \ldots, P_n$ are pictures. Then

$$\{P_1 \mid P_2 \mid \ldots \mid P_n\}$$

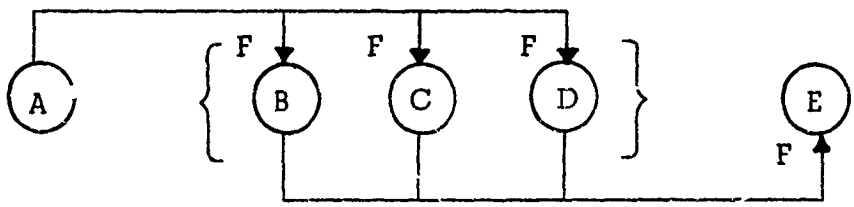
represents one of $P_1 - P_n$, and

$$\{P_1 \mid P_2 \mid \ldots \mid P_n\}_i \quad \text{for} \quad 1 \leq i \leq n$$

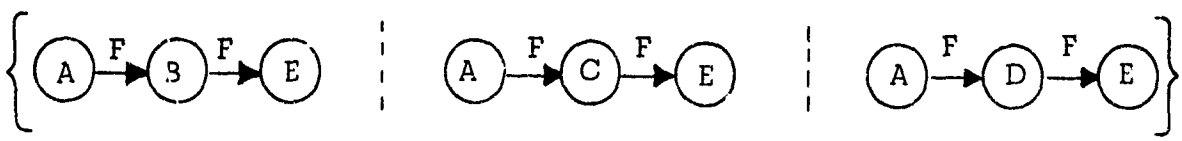
represents P_i. If the boundaries between each picture are clear, we drop the vertical dashed lines.

If the choice bracket represents a subtemplate, then an arc incident from outside the choice bracket must be made incident with each item in the bracket. We show this by splitting arcs.

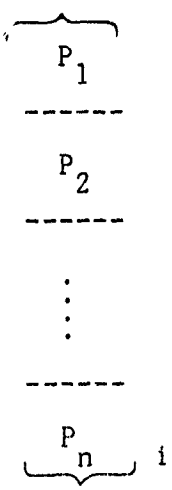
Thus



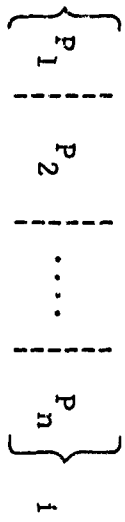
is equivalent to



A vertical bracket



is equivalent to the horizontal bracket



6.4 The (?) Node

We would like at times to indicate that when the current state is some particular node the command MOVEH A, for some A, will work but will move the current state to some unknown node. This could happen, for instance, if an index entry on a random access device were used before it was initialized.

To do this we introduce the "node" (?), which has the following properties:

1. ON NODE (?, ...) GOTO (S1, ...) is syntactically illegal, i.e. ? is not considered to be a node type.

2. ON NODE (T1, T2, . ., TN) GOTO (S1, S2, .., SN) . If this command is issued when the current state is (?) the result is uncertain. It might cause a branch to one of S1 - SN or it might cause an error.

3. MOVEH α, for any syntactically valid arc label α, may be a null operation or may cause an error.

4. WRITEW, READW and any other primitives which access the data portion of a node will either be null operations or produce an error.

7. IBM OS/360 ACCESS METHODS

7.1 Introduction

This is the first of three chapters wherein we attempt to describe three different data management systems using the data access representation as a framework. These descriptions should not be taken as formal definitions, nor as training manuals for data processing programmers. Rather they should be viewed as experiments, wherein the theoretical work described in previous chapters is measured against real-world problems.

The reader is also forewarned that chapters in this paper do not follow in chronological order of development. In fact this chapter, which describes IBM OS/360, was written quite early, long before the material in Chapter VI was invented. The reader will notice in this present chapter that no formal mechanism is given for deleting a member of a partitioned data set. It was precisely this difficulty that led to the creation of XFORM.

The source for this chapter is the IBM manual

> IBM System/360 Operating System
> Supervisor and Data Management Series
> Form C28-6646-2, November 1968

and to a lesser degree

> IBM System/360 Operating System
> Supervisor and Data Management Macro Instructions
> Form C28-6647-3, November 1968

7.2 Access Methods

In IBM terminology, an access method consists of two parts: data set organization and data access technique. A data set organization is a set of rules which defines a basic form for the data set graph. Any unit of data which can be fully represented by a graph corresponding to a data set organization is said to be a data set having that organization. It is possible for one unit of

data to be a data set having more than one data set organization. There are four types of data set organization: sequential, indexed sequential, direct, and partitioned.

A data access technique consists of a set of macro instructions and some special facilities which allow a program to efficiently transfer data between itself and a data set. There are two data access techniques: basic and queued. Basic access is asynchronous and unbuffered. The program must supply the access method with a block of data to write or a block of core in which to read data. A command to perform an activity merely initiates that activity. The program must issue a CHECK instruction to ensure that the operation is complete. Queued access uses internal buffering, look-ahead reading, and a string of output buffers to give the program the impression it is using I-O synchronously but which still allows I-O to overlap with program execution.

An access method consists of a data access technique and a data set organization. Theoretically all combinations are possible, but only the ones shown below have been implemented by IBM (as of Nov. 1968):

	data access technique	
data set organization	basic	queued
sequential	BSAM	QSAM
indexed sequential	BISAM	QISAM
direct	BDAM	
partitioned	BPAM	

7.3 Sequential Access Methods

OS/360 supports two sequential access methods, BSAM and QSAM. Both use the sequential data set organization. BSAM uses the basic data access technique, and QSAM uses the queued data access technique.

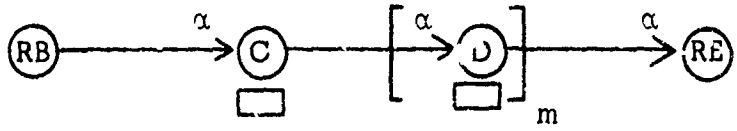
7.3.1 Sequential Data Set Organization

In order to define the sequential data set organization, we must define three templates: records, blocks, and data sets.

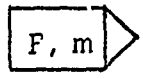
7.3.1.1 Records

There are three different templates for records: fixed length or format F, variable length or format V, and unformatted or format U. A particular data set may contain only one type of record.

7.3.1.1.1 Format F Records

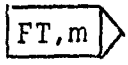


where m is the same for all records in the data set. We shall denote this template as

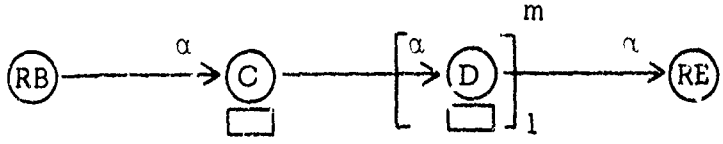


where | stands for (RB) and ⟩ stands for (RE) .

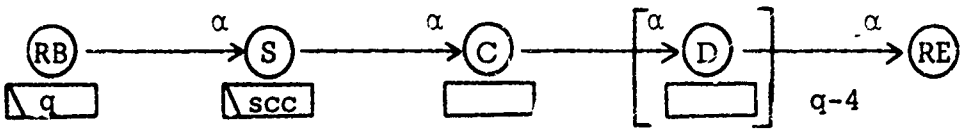
The type C node either contains a carriage control character or is ignored by the access method. The D type nodes contain the data, m words in all (bytes in OS/360).

The last record in a data set may be truncated, which we will designate as 

, which stands for

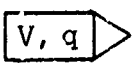


7.3.1.1.2 Format V Records

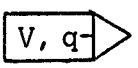


Here both q and scc may vary between records in a single data set. q is of course the record length. Nodes of type C, D, and RE are as before. Node RB now contains q in read-only mode. Node S contains an element called the Segment Control Code, determined by the value of scc. We shall draw a different macro for each value of scc:

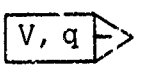
7.3.1.1.2.1 SCC = 0



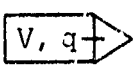
7.3.1.1.2.2 SCC = 1



7.3.1.1.2.3 SCC = 2

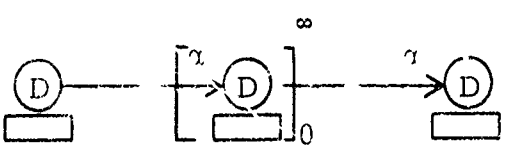


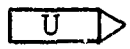
7.3.1.1.2.4 SCC = 3



7.3.1.1.3 Format U Records

or

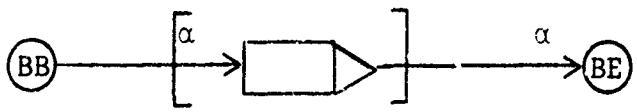


which we shall designate by 

, with | standing for the first data node and ⟩ standing for the last.

7.3.1.2 Blocks

There are three kinds of blocks: formats F, V, and U. All have the general shape

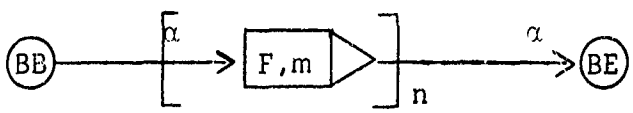


All block macros shall have the basic form

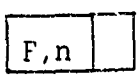
where [stands for BB and] stands for BE.

7.3.1.2.1 Format F

Format F blocks may contain only format F records, as follows:



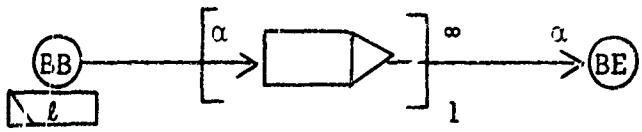
We shall designate this by

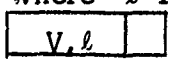


Note that specifying a value for the parameter n does not turn the block template into a graph, as the value of m is not yet fixed. We will not always specify in a template symbol all parameters necessary to turn the template into a graph. The reason should become obvious when we discuss format V blocks.

7.3.1.2.2 Format V

The basic form of a format V block is:

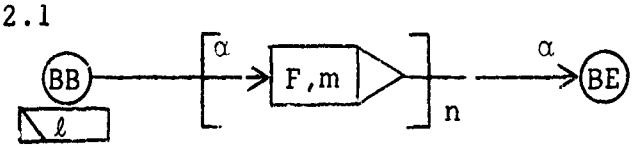


where ℓ is the physical length of the block. We shall designate this as 

. The number of records within a block is not explicitly known, and must be discovered by actually traversing the path from BB to BE.

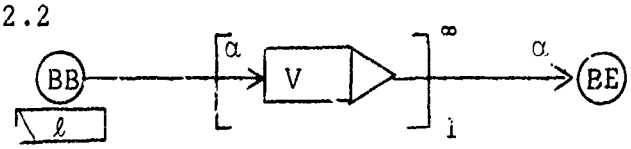
There are only certain forms the list of records may take:

7.3.1.2.2.1

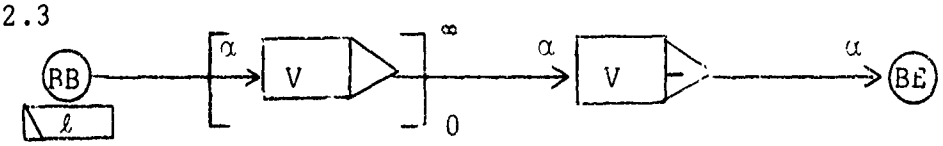


where $\ell = m \times n$, and $n > 0$.

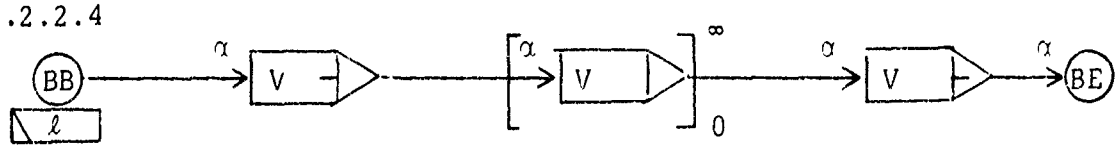
7.3.1.2.2.2



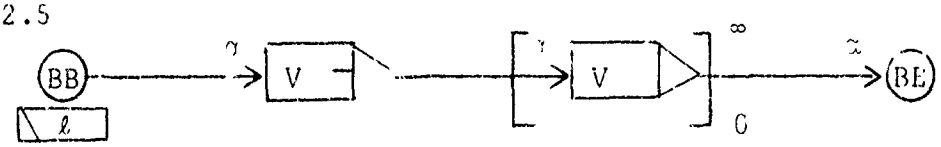
7.3.1.2.2.3



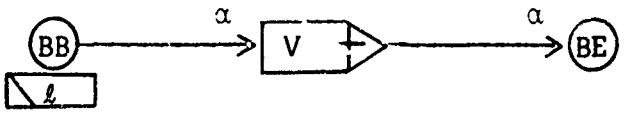
7.3.1.2.2.4



7.3.1.2.2.5



7.3.1.2.2.6



7.3.1.2.3 Format U Blocks

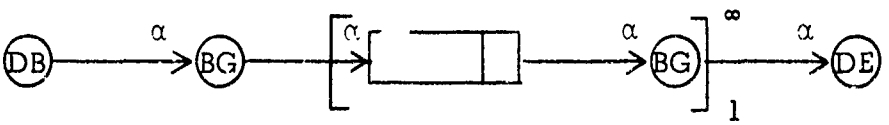
is the only allowable form. We shall denote it as 

.

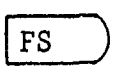
7.3.1.3 Data Sets

It is not all that clear from IBM publications just exactly what a data set is in general, or whether there even is a general, device independent concept of a data set. The following material must therefore be considered tentative and probably incomplete.

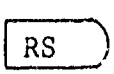
There are three radically different types of sequential data set. The most common one has the general form



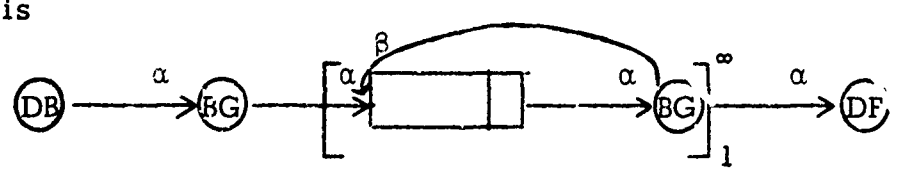
which we shall denote as



with | standing for DB and) standing for DE. In this form any block of the data set, once accessed, cannot be accessed again until the data set is closed. We shall call this forward sequential. This type of data set is supported on all devices. On direct access devices, once the read-write heads have been positioned to access a data block they are in position to access that block repeatedly. For this class of device there is a data set form that we shall call reflex sequential:

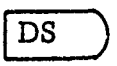


which is

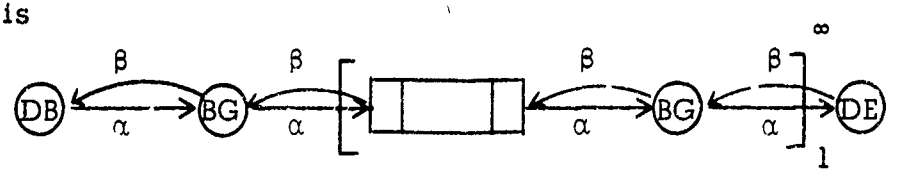


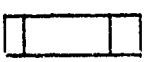
where the arc β allows one to REREAD or UPDATE the block most recently accessed.

Certain types of tape drive are capable of moving tape forward or backward while reading or writing. On such a tape unit it is possible to have a third data set form which we shall call doubly sequential



which is



where  stands for a block of data that can be accessed in either direction.

It should be clear by now that when considering problems of transferability one must treat these three forms of "sequential" data set to be radically different. From now on we shall confine our discussion to forward sequential data sets.

The data set template is set up at the time the data set is OPENed. It consists of the standard forward sequential data set template plus a set of parameters which refine this template. These parameters are stored in a table called the Data Control Block (DCB). It should be emphasized that these parameters are normally <u>not</u> sufficient to refine the template into a single graph. Information such as the number of blocks in the data set is generally not known. The DCB contains a number of fields, not all of which pertain to data set structure. Some of the pertinent fields are described below.

7.3.1.3.1 BLKSIZ = n

For format F blocks n is the number of words in a block. For format V or U blocks n is the maximum number of words in a block, including control words.

7.3.1.3.2 DEVD = code

This specifies the device type to be used and for certain devices special information such as recording density for a magnetic tape or stack number for a card punch.

7.3.1.3.3 LRECL = m or X

For format F records m is the number of data bytes in a record. If spanned records are allowed then m is the maximum length of a set of spanned record segments. X means that the maximum size is not specified.

7.3.1.3.4 RECFM = code

This is used to specify whether the records are to be format F, V, or U. If they are V then it also specifies whether or not spanned records are allowed.

7.3.2 Sequential Data Access Techniques

7.3.2.1 Queued Access

The queued access technique uses internal buffering which allows the user program to use synchronous data access commands and yet still have overlap between CPU processing and physical I-O processing.

OS/360 offers a variety of buffering schemes. All ensure that some finite pool of buffers is used to transfer an arbitrarily long stream of data records between the user program and the access method. We shall not, at this time, specify a standard for this buffering. We shall simply require that each system shall have at least one self-consistent buffering scheme in

exactly the same way for all devices that this access method supports. The specific buffering scheme used here is an imaginary one provided solely for logical concreteness.

7.3.2.1.1 Buffer Scheme

The access method provides three virtual registers: the buffer address register (BAR), the buffer size register (BSR), and the buffer pointer register (BPR). This triplet of registers defines the buffer currently in use: the BAR contains its address, the BSR contains its size, and the BPR points to the current word of interest within the buffer. The following commands are used to manipulate buffers:

7.3.2.1.1.1 GBUF

This gets a buffer of size LRECL (a parameter in the DCB), places its address in BAR, its length in BSR, and zero in BPR.

7.3.2.1.1.. MKBUF <addr>, <len>

This allows the user program to set up a block of storage as a buffer. It places <addr> in the BAR, <len> in the BSR, and zero in the BPR.

7.3.2.1.1.3 GVBUF

This returns the buffer defined by BAR and BSR to buffer pool.

7.3.2.1.1.4 READN

If BPR ≥ BSR the statement is in error. Otherwise the data attribute of the current node is placed into the memory location BAR + BPR and then BPR is incremented by one.

7.3.2.1.1.5 WRITEN

BPR must be < BSR. The word at memory location BAR + BPR is stored as the data attribute of the current node and then BPR is incremented by one.

7.3.2.1.2 More Notation

When the data set is opened under the queued access technique, the access mode must be set to either INPUT or OUTPUT. In INPUT mode the only allowable command is GET. In OUTPUT mode the only allowable command is PUT. The access mode, once set, cannot be altered until the data set is closed.

7.3.2.1.3 GET

On return BAR and BSR will contain a buffer containing the next (BPR + 1) words of data on the data set. The user program is expected eventually to use MKBUF and GVBUF to return this buffer to the pool. There are several forms of GET, depending on the block and record format.

7.3.2.1.3.1 Record F, Block F or V

```
GET     MACRO
        ONNODE      (DE, ELSE) GOTO (ENDFIL, ¢N1)
¢N1     ONNODE      (RB, ELSE) GOTO (¢N3, ¢N2)
¢N2     MOVEH       α
        GOTO        ¢N1
¢N3     MOVEH       α
        MOVEH       α
        GTBUF
¢N5     READN
        MOVEH       α
        ONNODE      (RE, ELSE) GOTO (¢N4, ¢N5)
¢N4     MOVEH       α
GET     MEND
```

ENDFIL is the location to which control is passed when an end of file is reached.

7.3.2.1.3.2 Record V, Block V

```
GET     MACRO
¢N1     ONNODE    (DE, ELSE) GOTO (ENDFIL, ¢N2)
¢N2     ONNODE    (RB, ELSE) GOTO (¢N4, ¢N3)
¢N3     MOVEH     α
        GOTO      ¢N2
¢N4     GTBUF
¢N8     MOVEH     α
        IF        (DATA (CURNODE) = 3), RESET SCCSW
        IF        (DATA (CURNODE) = 1), SET   SCCSW
        MOVEH     α
        MOVEH     α
¢N5     READN
        MOVEH     α
        ONNODE    (RE, ELSE) GOTO (¢N6, ¢N5)
¢N6     IF        (NOT SCCSW), GOTO ¢N7
¢N9     MOVEH     α
        ONNODE    (RB, ELSE) GOTO (¢N8, ¢N9)
¢N7     MOVEH     α
GET     MEND
```

Note that this form of GET assumes that the last record on a data set is not [V→] or [V→] . It also assumes that GETBUF returns a buffer long enough to hold any record or series of record segments. SCCSW is a binary switch in the access method. It may be SET true or RESET false.

7.3.2.1.3.3 Block U, Record U

```
GET     MACRO
        ONNODE    ( BB, ELSE) GOTO (¢N1, ENDFIL)
¢N1     MOVEH     α
        GTBUF
¢N2     READN
        MOVEH     α
        ONNODE    (BE, ELSE) GOTO (¢N3, ¢N2)
```

```
¢N3     MOVEH    α
        MOVEH    α
GET     MEND
```

7.3.2.1.4 PUT

PUT is used for output. It assumes that the user program has used MKBUF to set up BAR, BSR, and BPR with a block of memory which is to be written onto the data set. Upon return, this block of memory cannot be accessed by the program without risking an error. It is assumed that eventually the access method will use GVBUF to return this block of memory to the buffer pool. Again we will write different forms for PUT depending on the data set organization.

7.3.2.1.4.1 Block F, Record F

```
PUT     MACRO
¢N1     ONNODE   (RB, ELSE) GOTO (¢N3, ¢N2)
¢N2     MOVEH    α
        GOTO     ¢N1
¢N3     MOVEH    α
¢N6     MOVEH    α
        ONNODE   (RE, ELSE) GOTO (¢N5, ¢N4)
¢N4     WRITEN
        GOTO     ¢N6
¢N5     MOVEH    α
PUT     MEND
```

7.3.2.1.4.2 Block V, Record V (assume spanned records)

```
PUT     MACRO
¢N1     ONNODE   (RB, ELSE) GOTO (¢N3, ¢N2)
¢N2     MOVEH    α
        GOTO     ¢N1
¢N3     MOVEH    α
¢N4     MOVEH    α
        ONNODE   (RE, ELSE) GOTO (¢N6, ¢N5)
```

```
¢N5        IF          (BSR ≤ BPR) GOTO (¢N7)
           WRITEN
           GOTO        ¢N4
¢N6        IF          (BSR ≤ BPR) GOTO ¢N7
           GOTO        ¢N1
¢N7        MOVEH       α
PUT        MEND
```

7.3.2.1.4.3 Block U, Record U

```
PUT        MACRO
¢N1        ONNODE      (BB, ELSE) GOTO (¢N3, ¢N2)
¢N2        MOVEH       α
           GOTO        ¢N1
¢N3        MOVEH       α
           ONNODE      (BE, ELSE) GOTO ¢N5, ¢N4)
¢N4        WRITEN
           GOTO        ¢N3
¢N5        MOVEH       α
PUT        MEND
```

7.3.2.2 Basic Access

In basic access mode the allowable commands are READ, WRITE, and CHECK. Execution of a READ or WRITE does not cause an immediate transfer of data. Instead it creates a promise of such a transfer, called a Data Event Control Block (DECB). This DECB is pushed onto the bottom of a stack which we shall call the Pending Command Stack (PCS). The access method, at its leisure, will execute the DECB on the top of the stack. When it is done it will mark the DECB complete and remove it from the stack. Before the program may use the data obtained from a READ DECB or may reuse the buffer space in a WRITE DECB it must issue a CHECK instruction for that DECB. Control will not be returned from a CHECK instruction until the DECB in question, and all DECB's higher than it on the PCS, have been executed and removed from the stack.

It is an interesting question in just what "state" the access method is when the PCS is not empty. If we assume that the program always CHECKs before using any core memory area mentioned in a DECB then we may consider the access method "state" to be the state it would be in after it had emptied the PCS. Normally we may accept this definition of "state" and for all intents and purposes ignore the existence of the PCS. If an error occurs, however, the "state" of interest will be the real access method state. The treatment of I-O errors in an asynchronous access method is a complex problem that will not be considered at the moment. It should also be noted that the queued access mode has the same problem.

The allowable commands in the basic access mode are READ, WRITE, and CHECK. READ and WRITE deal not with records, but with blocks. Blocks are treated as if they were either format F or format V. Records within blocks are ignored, and record control information is treated as if it were data.

In the following READ and WRITE macros it should be remembered that the READs and WRITEs are performed by the access method, not the user's code. The true READ and WRITE macros for the user program merely create DECBs and push them onto the PCS.

7.3.2.2.1 READ

7.3.2.2.1.1 Format F

```
READ      MACRO
¢N1       ONNODE     (BB, DE, ELSE) GOTO (¢N3. ENDFIL, ¢N2)
¢N2       MOVEH      α
          GOTO       ¢N1
¢N3       MOVEH      α
          ONNODE     (BE, ELSE) GOTO (¢N5, ¢N4)
¢N4       IF         (ACCESS (NODE) = NR) GOTO ¢N3
          READN
          GOTO       ¢N3
¢N5       MOVEH      α
READ      MEND
```

7.3.2.2.1.2 Format V

```
READ      MACRO
¢N1       ONNODE     (BB, DE, ELSE) GOTO (¢N3, ENDFIL, ¢N2)
¢N2       MOVEH      α
          GOTO       ¢N1
¢N3       MOVEH      α
          MOVEH      α
          ONNODE     (BE, ELSE) GOTO (¢N5, ¢N4)
¢N4       IF         (ACCESS (NODE) = NR) GOTO ¢N3
          REALN
          GOTO       ¢N3
¢N5       MOVEH      α
READ      MEND
```

7.3.2.2.2 WRITE

7.3.2.2.2.1 Format F

```
WRITE     MACRO
¢N1       ONNODE     (BB, ELSE) GOTO (¢N3, ¢N2)
¢N2       MOVEH      α
          GOTO       ¢N1
¢N3       MOVEH      α
          ONNODE     (BE, ELSE) GOTO (¢N5, ¢N4)
¢N4       WRITEN
          GOTO       ¢N3
¢N5       MOVEH      α
WRITE     MEND
```

7.3.2.2.2.2 Format V

```
WRITE    MACRO
¢N1      ONNODE     (BB, ELSE) GOTO (¢N3, ¢N2)
¢N2      MOVEH      α
         GOTO       ¢N1
¢N3      MOVEH      α
¢N4      MOVEH      α
         ONNODE     (BE, ELSE) GOTO (¢N6, ¢N5)
¢N5      WRITEN
         GOTO       ¢N4
¢N6      MOVEH      α
WRITE    MEND
```

7.4 Partitioned Access Methods

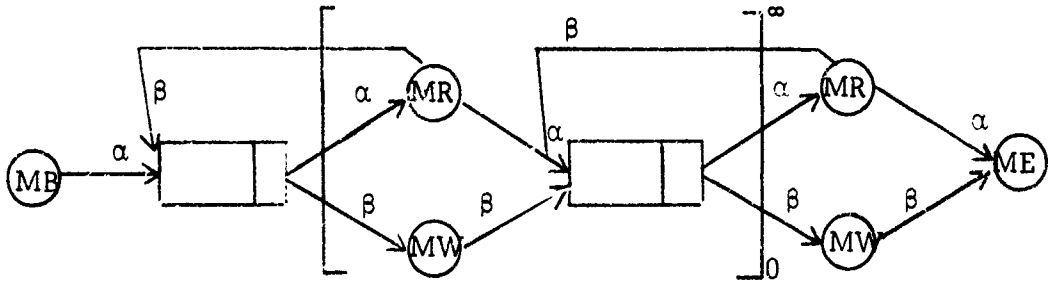
IBM claims that a partitioned data set is simply a collection of members, each of which can have any legal sequential data set format. For the purpose of this discussion we will assume this to be true.

A partitioned access method is basically a sequential access method with the addition of two new command macros, FIND and STOW. These have approximately the same relationship to members of a partitioned data set as OPEN and CLOSE have to sequential data sets. Thus in a sense we have nothing new, as we agreed before to consider an algorithm as starting when the data set is opened and stopping when it is closed. With a partitioned data set (PDS), however, FIND and STOW are so much faster than OPEN and CLOSE that we must consider them as being embedded in the algorithm. Thus our model of a partitioned access method must explicitly include machinery to allow the control taken to visit more than one member of the data set currently open.

In order to do this, we must have an explicit representation of the collection of member names and of the process by which these names are located. This representation must also accurately model the process by which members are added, deleted, or altered.

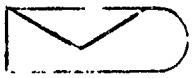
7.4.1 Member

As far as we can tell from IBM documentation, a member can have any sequential data set form, but with one extra twist. If a WRITE macro immediately follows a READ macro, something which is possible only in UPDATE mode, the block written will be the block just read. Thus we must have the following form for a member:



where we will use β for writing and α for reading.

We abbreviate this as:

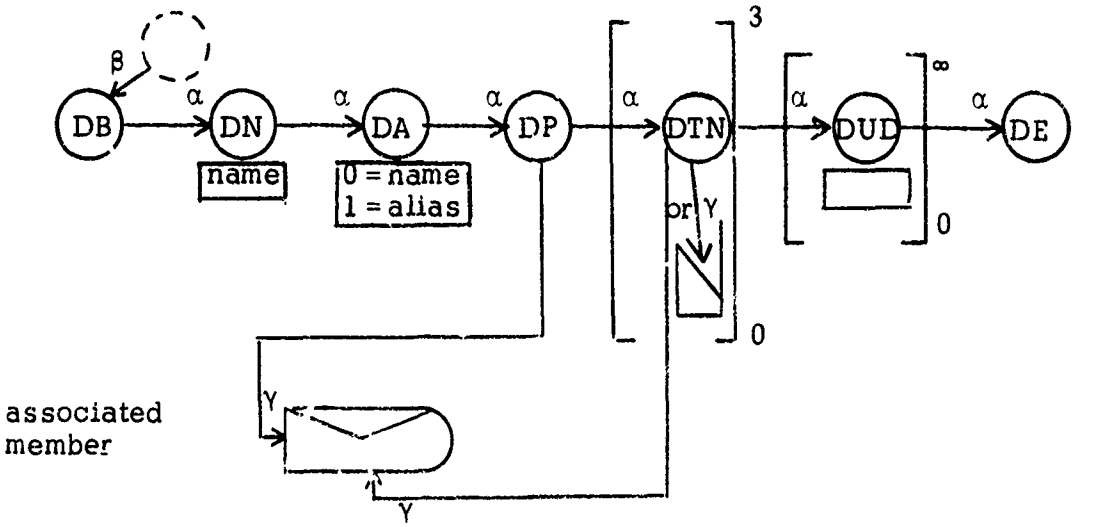


with | representing MB and) representing ME.

7.4.2 Directory Entry

For each member in a partitioned data set there is at least one directory entry. Each entry for a member gives that member a name. One entry is flagged as the official member name and the others are flagged as aliases.

Each directory entry contains a pointer to the beginning of the member it defines. It also contains an optional user data area. This area can contain arbitrary information, but usually contains pointers to locations inside the member. A directory entry can have the following form:



where

DB is the beginning of the directory entry

DN contains the name, or alias, of a member

DA contains 0 if DN contains a name and contains 1 if DN contains an alias

DP points to the data set member

DTN points either to a note list or to the beginning of a block in the data set member

DUD contains arbitrary data

DE is the end of the directory entry.

We shall use the abbreviation

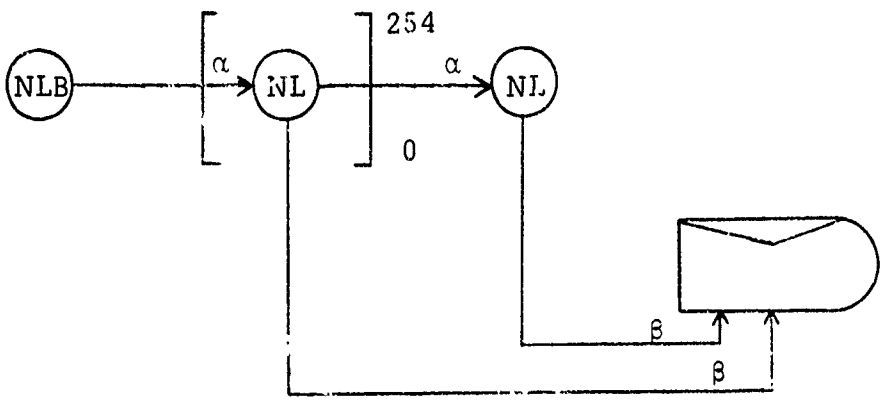
We have taken some liberties with our previous notational scheme here. The arc labeled α in

represents not just a particular arc but rather the possibility of drawing an arc from any of the DTN nodes to any BB node in the data set member. We shall call this a representative arc. Furthermore, the portion of the macro symbol from which the arc emanates represents not just one DTN node but all DTN nodes in that directory entry. We call this a representative node.

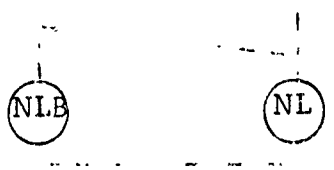
We shall make some use of representative nodes and arcs in the following discussion. We shall not attempt to formalize rules for their use, however. It is hoped that the context of each individual use will make the meaning of that use clear.

7.4.3 Note List

A note list is simply a list of pointers to blocks within a member of a data set. Its template is:



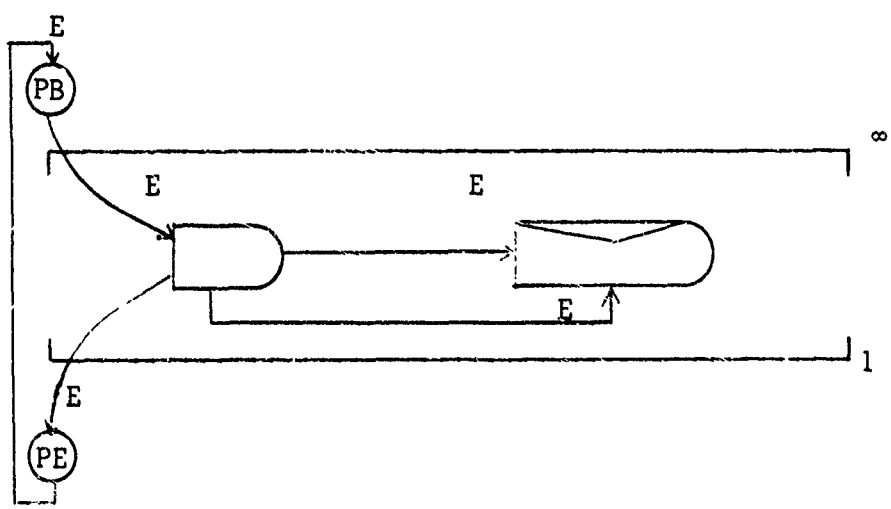
with symbol  and equivalence



A DTN node in a directory entry may point either to a block in the member or to a note list, which itself contains pointers to blocks in the member. Thus we could in one sense subsume note lists into the directory. There is one important practical problem, however, which makes it worthwhile to include an explicit representation for note lists. A DTN node is part of the directory entry and has been allocated space in the same block as all other nodes in this directory entry; a note list is a separate record somewhere else on the list. It requires an extra read and perhaps an extra disk head seek to access a note list. The overhead involved in doing this may well be a crucial factor in practical problems of program transferability.

7.4.4 Partitioned Data Set

A partitioned data set consists of a directory and a set of members, as follows:



subject to the following variations and/or restrictions:

The directory entries are collated by the value of the DN node, i.e., the names are sorted alphabetically.

Some of the members may be identical, i.e., more than one directory entry may point to a particular member.

Any arc labeled E going from a DTN node to a BB node may be replaced by

7.4.5 Space Allocation

One feature of the graph representation for a partitioned data set is that it hides almost completely the process of space allocation on a physical volume. This is convenient, as the access method is so constructed as to shield the user quite thoroughly from any specific considerations of space allocation. This gives the access method quite a bit of freedom to adapt to different devices and different data set histories without requiring changes to any user program.

Space allocation in a partitioned access method is quite simple. The size of a data set is fixed when the data set is first created, and cannot be changed without recopying. The space can be thought of as a block of contiguous words. A fixed sized area at the front of the data set is reserved for the directory. The size of this area limits the total number of member names and aliases and the total amount of user data in the directory. As long as this limit is not reached, the access method will keep the directory properly sorted and up to date.

The rest of the data set area is used to store members and note lists. A member is stored as a set of contiguous words. The first member defined is stored directly above the space allocated for the directory; each succeeding member is stored immediately after the member that was defined just prior to it. If a member is deleted and it is not the member most recently defined then the space allocated to it becomes unusable; no attempt is made to move other members down to fill the space created.

7.4.6 FIND

FIND is given a member name or alias. It will search the data set directory for an entry of that name. If it does not find that name it will return an error-code to that effect. If it does find the name, on return the access method token will be positioned on the MB node of the appropriate member.

7.4.7 STOW

STOW causes an entry in the directory to be added, deleted, or changed in name. The specific details of the method of using STOW are quite implementation-dependent. We will give here a more abstract, and hopefully more understandable presentation.

7.4.7.1 Add a Name

The directory is sorted alphabetically by name. A new directory entry is added between the name ordering just before it and the name ordering just after it.

7.4.7.2 Change a Name

The directory entry corresponding to the old name is removed and an entry for the new name made at the proper place in the directory.

7.4.7.3 Delete a Name

The directory entry for this name is removed from the directory. If this is the only name of a particular member, then that member will thereby become unreachable.

7.4.7.4 Replace a Member

If the name given is not already in the directory, it is added. Otherwise, the DP pointer in the directory entry is made to point to a new member. This may make the old member unreachable.

7.5 Indexed Sequential Access Methods

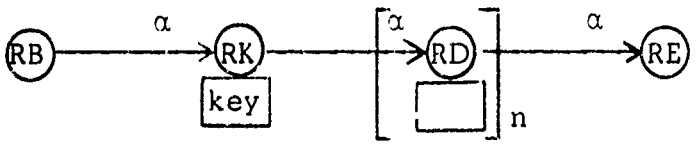
The primary feature of indexed sequential data sets is that each data record has a key. Records are ordered more or less sequentially on the key value. Records may be accessed sequentially using the queued access technique or they may be accessed directly using the basic access technique. A record may be removed, added, or changed in size by using the basic access technique. The access method will automatically move other records as necessary to recover any gaps left by the removal or reduction in size of a record.

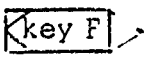
Again IBM documentation is not too clear, but it appears that the basic unit of information is the keyed record and the block, if it exists at all, is used only internally by the access methods. We shall assume here that blocks do not exist.

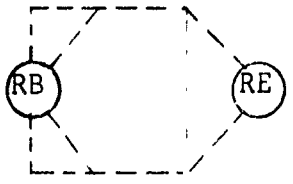
7.5.1 Records

7.5.1.1 Format 1

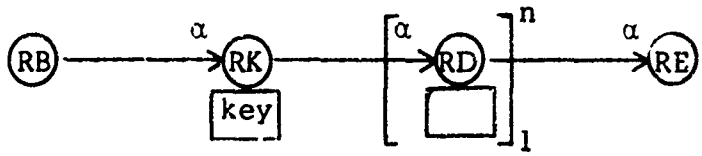
Suppose LRECL = n



With symbol  and equivalence

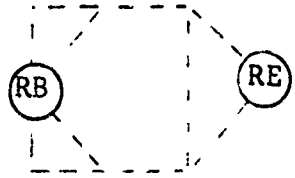


7.5.1.2 Format V



with symbol 

and equivalence.



7.5.2 Tracks and Cylinders

While the user is to some degree shielded from the fact, tracks and cylinders play a fundamental role in the organization of indexed sequential data sets. Their precise meaning varies from device to device, but roughly speaking a track is a fixed number of data words which can be accessed sequentially and a cylinder is a fixed number of records such that the overhead involved in switching between two tracks in the same cylinder is less than the overhead involved in switching between cylinders. The important fact here is that the number of data words on a track and the number of tracks per cylinder is fixed for a given device and varies from device to device.

To expedite locating a record with a given arbitrary key several levels of indexing are provided. At the lowest level is the track index, of which there is one per cylinder. For every prime data track (to be defined later) there are two entries in the track index. The first entry is the highest value key on that track. The second is the highest value key of any entry assigned to that track but stored in the overflow area.

The data set also contains a cylinder index with one entry per cylinder, giving the highest key value of any record on that cylinder.

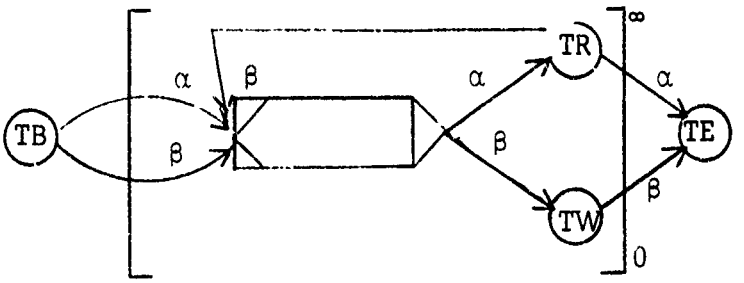
If the cylinder index is so large it covers more than one track then a master index is created. There is one entry in the master index for each track of the cylinder index and this entry gives the highest value found on that track.

For extremely large data sets it is possible to create even higher level indexes. As they add nothing conceptually new we will not model them here.

7.5.3 Tracks

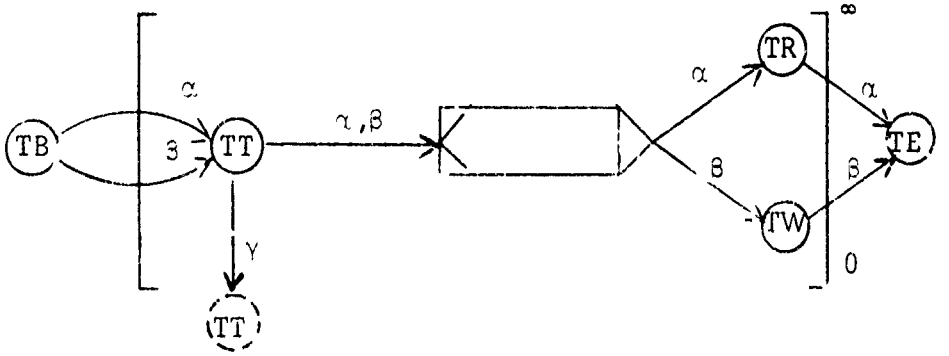
There are two types of tracks in an indexed sequential data set: prime data and overflow.

A prime data track is a sequence of records in ascending order by key value:



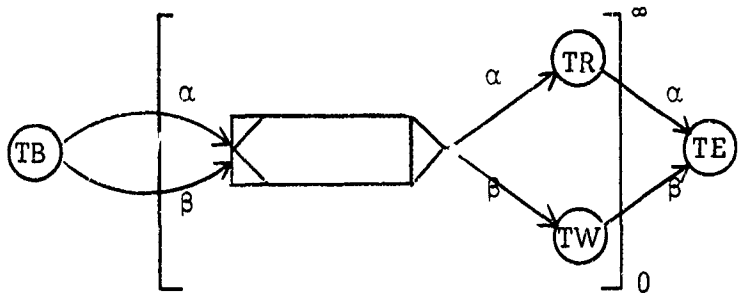
The total length of all records on the track must not exceed some limit set by the device, say TRKLEN.

Overflow tracks are much more complex. At the level of detail we have been using up to now an overflow track has the following template:

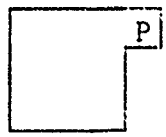


where (TT) is representative of any TT node on any overflow track on this cylinder, or the special node (NIL) . The TT nodes are used to string overflow records into lists.

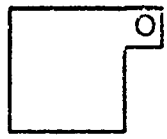
The indexed sequential access methods contain some relatively complex code which uses all of the overflow tracks to maintain one overflow record string for each prime data track on the cylinder. The specific algorithms used to allocate space and maintain the strings are invisible to the user and are not officially specified. Thus we cannot and should not include in our model a specific overflow track management scheme. We must replace the overflow tracks by overflow lists.



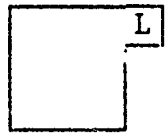
This is exactly the same template as for prime data tracks. Again the keys are in ascending order. There is no fixed maximum size, however, as this is dependent on the other lists. The arcs, while logically consistent, now represent rather complex operations. Thus while it is possible to assign average costs to these arcs, great variations from these averages can be expected when the access method is actually run. We shall denote prime data tracks as



overflow tracks as

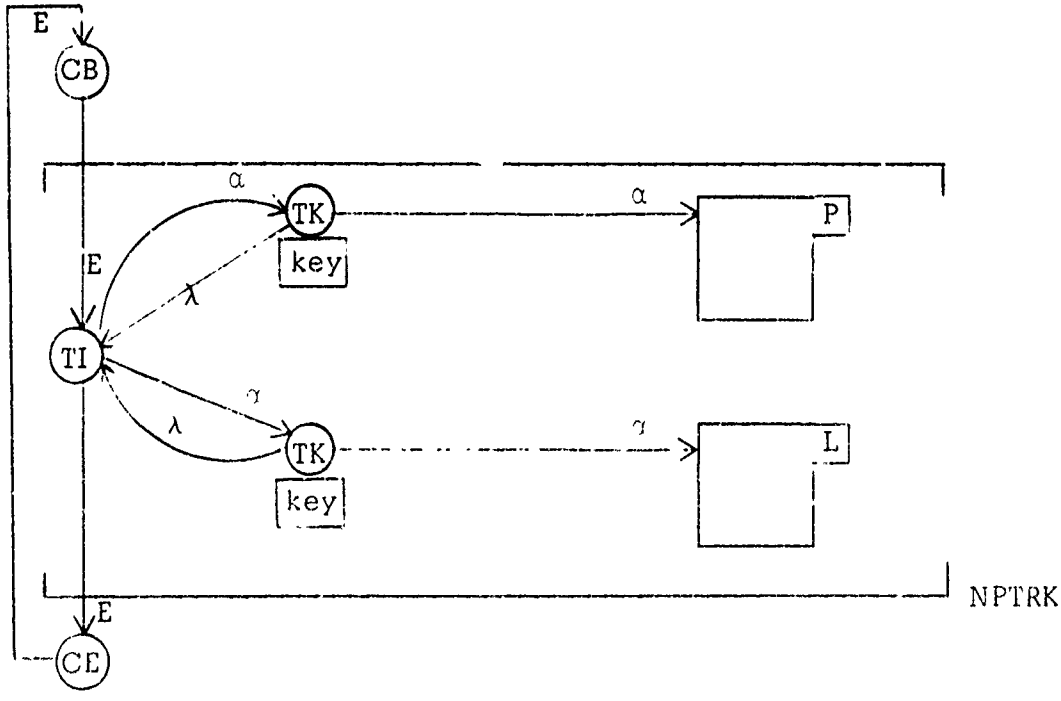


and overflow lists as



7.5.4 Cylinders

The physical tracks on a cylinder are divided into three areas. The first track or so is used to store the track index. The bulk of the tracks are prime data tracks. The remaining tracks are overflow tracks. If NPTRK is the number of prime data tracks, a cylinder may be represented by



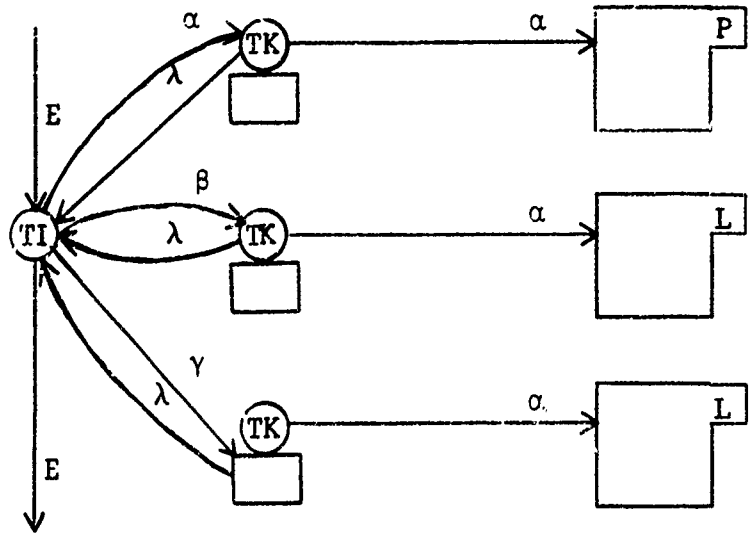
The value of a TK node is the highest value key in the attached track or overflow list. The overflow key must be higher in value than the prime key. Furthermore, the prime key of one prime track must be higher than the overflow key of the preceeding prime track. Any [P] or [L] may be replaced by the node (NIL), signifying an empty list or track. The TK node for an empty list or track has value -1, one less than the smallest possible key value.

We shall denote a cylinder as

with equivalence

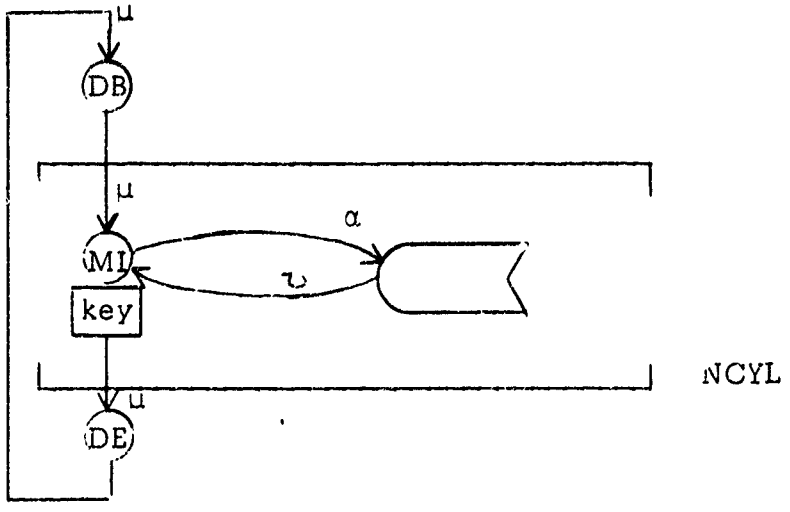
7.5.5 Data Sets

An indexed sequential data set consists of one or more levels of master index (not modelled here), one or more cylinders of data, and optionally one or more cylinders of independent overflow area. The independent overflow area is used to contain overflows from cylinders whose overflow tracks are full. Conceptually it adds an extra overflow list to each prime data track:



As it adds nothing new conceptually, we shall not model it here.

An indexed sequential data set has the form



where NYCL is the number of prime data cylinders. The value of an MI node is the highest key found on the associated cylinder, or -1 if the cylinder is empty. Successive non-negative MI values must be in ascending order.

7.5.6 Queued Indexed Sequential Access Method (QISAM)

Although we will present here QISAM and BISAM separately, neither is really a self-sufficient access method. To create, access, and maintain an indexed sequential data set it is usually necessary to use a mixture of both access methods. We quote from the IBM System/360 Operating System: Supervisor and Data Management Services manual (C28-6646-2) p. 137:

"Although the queued and basic access techniques can be used to process an indexed sequential data set, each has separate and distinct functions. The queued access technique must be used to create the data set. It can also be used to process or update the records. Only the basic access technique can be used to insert new records between records already in the data set. It too can be used to read the data set or update records. However, you may add new records to the high key end of the data set using the queued access method."

While it isn't all that clear exactly what happens, it appears that the queued access technique writes records only in prime data tracks while the basic access technique always causes records to be written into the overflow tracks, either directly or by displacement from the prime data tracks. Thus if the basic access technique were used to create a data set, all of the records would be written into the overflow tracks.

The commands available in the queued access technique are GET, PUT, SETL, and ESETL.

7.5.6.1 SETL, ESETL

SETL allows the user to select a point other than the beginning of the data set to start sequential retrieval of records. It can only be used while reading. SETL takes as argument a key, a key prefix, or an absolute track

address. A key prefix has fewer characters than a key. If a key prefix is given, ESETL will find the first key in the data set with the same initial characters as the key prefix. After a SETL, subsequent GET instructions will retrieve records sequentially starting from the record selected. If SETL is to be called more than once, all but the first must be preceded by a call to ESETL. ESETL destroys the look-ahead buffers and other machinery set up by SETL. It places the data set in some sort of limbo which the manual doesn't bother to define.

7.5.6.2 GET, PUT, PUTX

It is possible to access an indexed sequential data set in input, output, or update mode. In input mode, only GET is allowed. In output mode, only PUT is allowed. In update mode GET and PUTX are allowed. PUTX will cause the record referred to in the most recent GET statement to be replaced by the record given to the PUTX statement.

7.5.7 Basic Indexed Sequential Access Method (BISAM)

BISAM is set up in such a manner as to simulate direct access of records, using the record key as address. It is possible in update mode to WRITE the record most recently READ, but other than this the system makes no use of information about the current record in determining the next record. The commands possible under BISAM and READ, WRITE, and CHECK. The IBM Supervisor and Data Management Services manual (C28-6646-2) contains (p. 102) some self-contradictory nastiness about using WAIT instead of CHECK, but we will ignore this.

7.5.7.1 READ

READ takes as argument a key, and returns the record having that key. There are two modes, K and KU. In mode K nothing additional happens, but in mode KU the physical address of the record is placed in the DECB. A subsequent WRITE command using the same DECB will cause the record to be updated in place.

7.5.7.2 WRITE

While READ causes nothing more than the access of a record, a WRITE command has much more far reaching effects. There are two modes for WRITE, K and KN. Mode K is used to update in place, and causes no changes in data set structure beyond the bounds of the current record. It can have a later effect, however, as the record update may have included the insertion of a delete code. This is a byte of all ones (X'FF') as the first character of the record. Such a record may be deleted if a subsequent WRITE in KN mode accesses that track.

Mode K requests require that the new record replace an existing record of the same key. Mode KN requires that the key of the record to be written not already exist in the data set. If the key of the new record is higher than any key presently in the data set, the record is added to the overflow list of the last prime track currently used. In all other cases the new record must be placed between two already existing records. If the new record must go on an overflow list, it is simply written there and the list pointers adjusted accordingly to string the new record into the proper location on the overflow list. If the record must go between two records already on the same prime track, however, space must be made on the prime track for it. This is done by moving all records on that prime track with higher keys than the new key up a sufficient distance to accomodate the new record. This will in general force one or more records off the end of the prime track and onto the overflow track. The access method will do this automatically. If a record which is forced off the end of a prime track contains a delete code, it will not be written onto the overflow list but will simply disappear.

8. CDC SCOPE ACCESS METHODS

The Control Data Corporation SCOPE operating system for the CDC 6400, 6500, and 6600 computers offers a complete access method for secondary storage. Compared to the IBM access methods, it offers a smaller number of file structures, but a larger number of macros for access. There is no distinction between blocks and records. The atomic unit of data is a six bit character. The smallest unit of transaction with external devices is the physical record unit or PRU. This is a fixed length string of characters, with the length determined by the characteristics of the specific device used. A logical record consists of one or more physical record units, with the last PRU truncated or zero length. A truncated or zero length PRU contains a flag signalling the end of data. Depending on the flag used, a truncated or zero-length PRU signifies the end of a logical record, the end of a file, or the end of a volume.

Compared to OS/360, the SCOPE system offers a much simpler buffering scheme. The sole interface between the program and the I-O device is a circular buffer. This is a block of contiguous memory within the user program which is treated as if the first location immediately followed the last, forming a ring.

SCOPE supports only two data structures, corresponding roughly to OS/360's sequential and indexed sequential. As with OS/360, sequential files may be accessed either forward or backward, if the I-O device allows it.

This description was originally written using a manual for SCOPE version 3.1.6 of November 1969. It was later updated to conform with SCOPE version 3.3 of February 1972. Our description does not include SCOPE Indexed Sequential, which was added between versions 1 and 3 and which is apparently a separate program package which uses ordinary SCOPE data management as a base. It would be quite interesting to compare it with IBM Indexed Sequential.

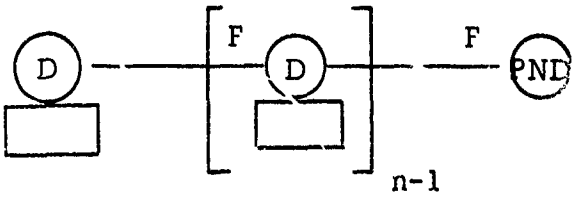
8.1 SCOPE Access Method Elements

8.1.1 Character

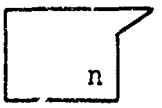
There are two basic data forms in SCOPE: display code and binary. The smallest unit of data is six bits wide and corresponds to one display code character or two octal digits. We will denote a character by

8.1.2 Physical Record Unit (PRU)

This device dependent quantity is the smallest unit of transaction between the access method and the device. For any given device there is an n such that



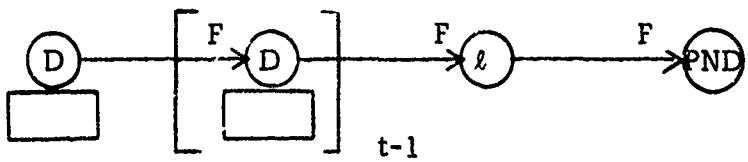
represents a PRU for that device. We assign this the macro



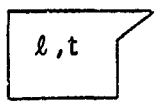
where [represents the first node and ⌉ the last node.

8.1.3 Truncated PRU

Less than n words of data may be stored in a truncated PRU. This contains t words of data, where $1 \leq t \leq n-8$, and an end flag:



ℓ may be any one of 16 types, corresponding to levels 0-15: L0, L1, L2, ..., L15. We give this the macro symbol

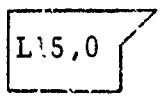


8.1.4 Zero Length PRU

A zero length PRU contains only the end flag. It is represented by

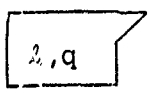
and has macro 

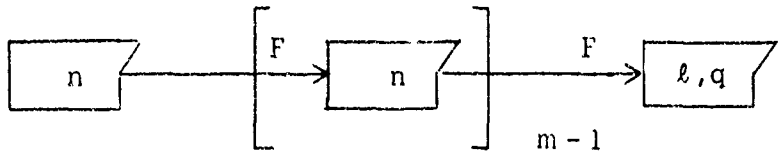
. Truncated and zero length PRU's are used to delimit logical records and files. Node type L15 is reserved for an end-of-file mark:



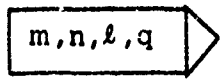
8.1.5 Logical Record (SCOPE Standard)

A logical record consists of zero or more full PRU's followed by a truncated or zero length PRU:

for m = 0 

or for m > 0 

This is given the macro symbol

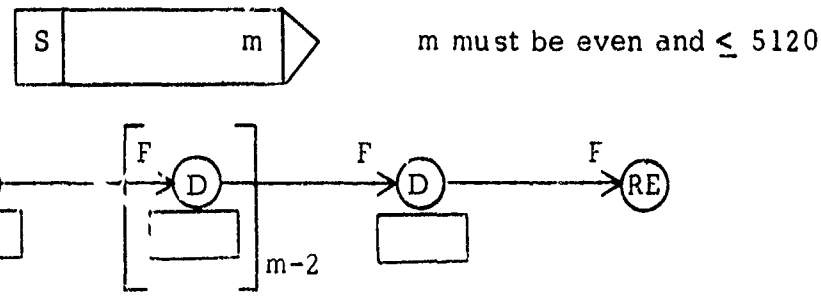


ℓ may not be L15. If ℓ = L14 the record will receive special treatment by the checkpoint dump program of SCOPE.

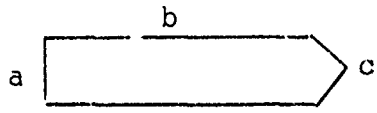
8.1.6 Nonstandard Tapes

SCOPE offers some facilities to enable a nonstandard tape to be accessed. This allows tapes to be sent to or received from other operating systems, telemetry equipment, and the like. It also provides compatibility with earlier versions of SCOPE. We will describe these special tape formats, but will not attempt to make programs using these formats transferable, at least not at the moment.

8.1.6.1 Type S (Stranger) Tape



with the following equivalence



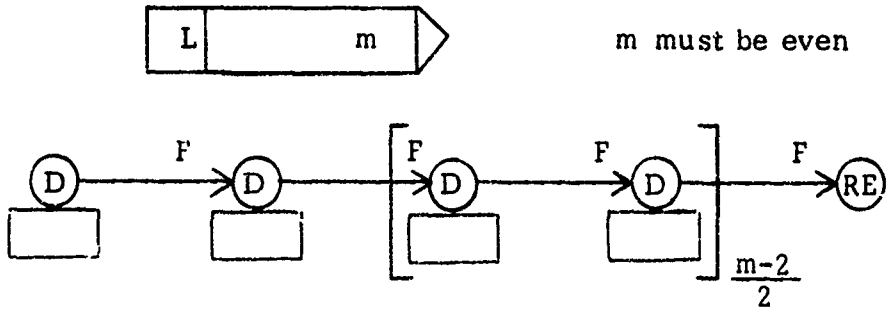
where

a is the first type D node

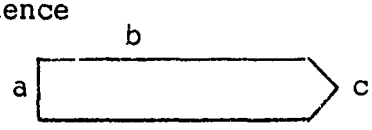
b is any node

c is the RE node

8.1.6.2 Type L (Long Record Stranger) Tape



with equivalence



where

a is the first D node

b is any node

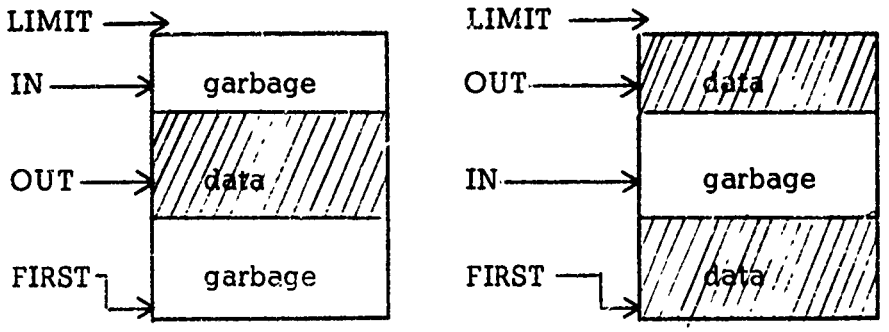
c is the RE node

8.1.7 Circular Buffer

This is a block of contiguous memory locations within the user program's address space. It is described by four registers: FIRST, OUT, IN, and LIMIT. The buffer begins at the address in FIRST. LIMIT is the first address beyond the end of the buffer. The two registers OUT and IN partition the buffer into two regions, data and garbage. OUT and IN satisfy the following constraints:

$$FIRST \leq OUT < LIMIT$$
$$FIRST \leq IN < LIMIT$$

If OUT = IN then the whole buffer is garbage. If OUT < IN then the region from OUT to IN -1 is data and the rest is garbage. If OUT > IN then the region from OUT to LIMIT -1 and the region from FIRST to IN -1 contain data and the rest is garbage. These relations may be described graphically as follows:



The buffer size m = (LIMIT-FIRST) must be sufficient to contain at least one full PRU and preferably several.

8.1.8 Write A PRU

We will use a template transform to describe the process of writing PRU's. The commands will be

XFORM FP

for full PRU's of length n, and

XFORM TP (ℓ, q) $0 \leq \ell \leq 15$, $0 \leq q \leq$ PRUSZ

The covering templates for these transformations will be specified as the need arises. Any covering template, however, must have in its name a positive integer valued parameter PRUSZ. It should be noted that the transform

TP(ℓ, q)

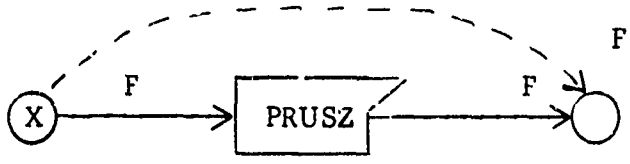
is not a single transform but rather a shorthand for 16 (PRUSZ + 1) functions.

FP or any one of the TP(ℓ, q) is a triplet of functions. The first and third functions specify which parts of the data structure remain invariant, and the second function defines the part that changes. If we let FP = (FP_1, FP_2, FP_3) and TP(ℓ, q) = $(TPLQ_1, TPLQ_2, TPLQ_3)$ we may define the transformations as follows:

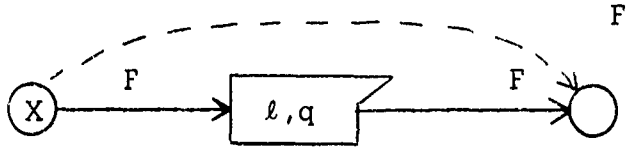
8.1.8.1 FP_2, $TPLQ_2$

The current state must have an exit arc labelled F. While the transform doesn't require it, the current state should be of type RE, ER, or PND and the state reached from the current state along the F arc, which we will call the F successor, should be of type D or L0, L1, ..., L15.

FP_2 breaks the F arc between the current state and its F successor and puts a full PRU in its place. We will describe this graphically as follows:



Similarly $TPLQ_2$ inserts a truncated or zero length PRU of level L:



8.1.8.2 FP_1, FP_3, $TPLQ_1$, $TPLQ_3$

As should be evident from the above, all nodes in any graph of the covering template are passive, and the only structural change consists of the addition of a PRU between the current node and its F successor.

8.1.9 XFMTPL ℓ

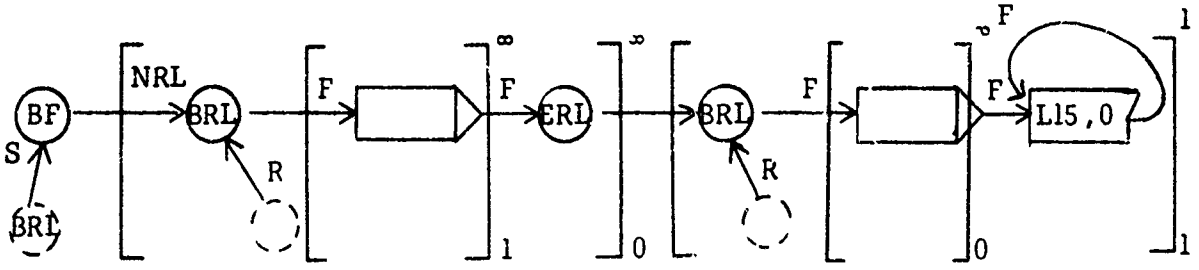
This is a special form of XFORM TPLQ keyed to the circular buffer. Let q = ((IN-OUT) mod m), the number of data words in the buffer. If q> PRUSZ -6 there is an error. Otherwise XFMTPL selects and executes

XFORM TP(ℓ,q)

8.2 Forward Sequential File Structure

There are three basic file structures allowed in SCOPE: forward sequential, doubly sequential, and indexed. SCOPE does not explicitly distinguish between forward and doubly sequential but, for reasons stated before, we feel it is important to do so.

A forward sequential file has the following form:

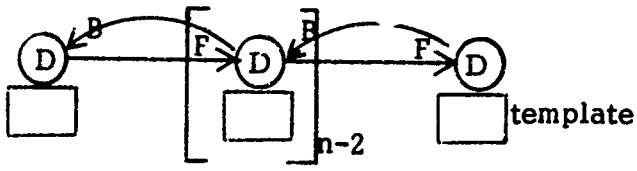


When the file is opened the current node will be the first node of the first record of the first reel, i.e., the node reached from node BF by executing MOVEH NRL, MOVEH F. Note that there is at least one reel, that the end of file indicator is a zero length PRU of level 5, that the head refuses to move past the end of file, and that rewind applies only to the current reel and can be performed at any time.

8.3 Doubly Sequential File Structure

The SCOPE system does not appear to provide any facilities for reading or writing backward. On devices capable of reading or writing backward, however, SCOPE does provide the ability to space backward a number of PRU's, a number of logical records, or to the first logical record with level equal to or higher than a particular level number. This requires that SCOPE read the file backwards in order to find logical record ends and level numbers. Thus we must include in our file structure a mechanism for reading backward, even if the user program is not allowed to employ it. Thus the doubly sequential form is as follows:

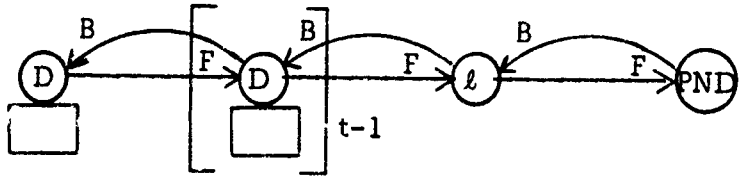
8.3.1 PRU



template

macro

8.3.2 Truncated PRU

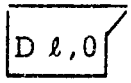


template

macro

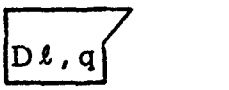
8.3.3 Zero Length PRU

template

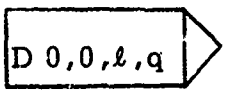


macro

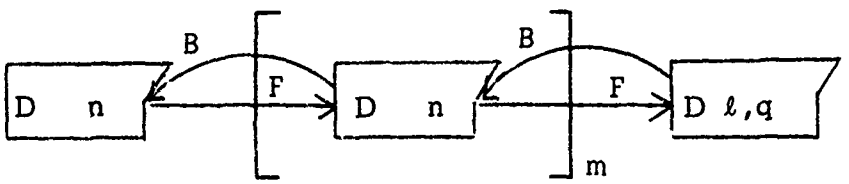
8.3.4 Logical Record



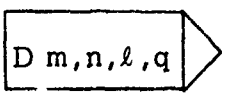
template



macro

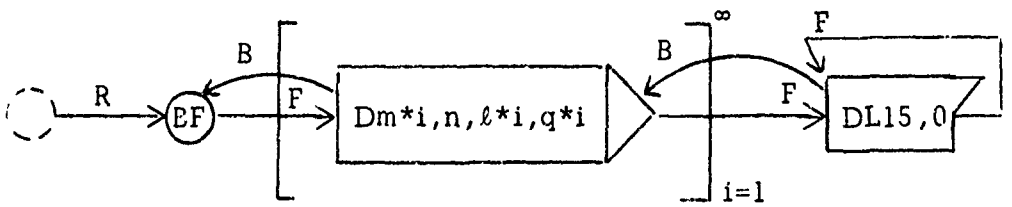


template



macro

8.3.5 File



8.4 Random Access File Structure

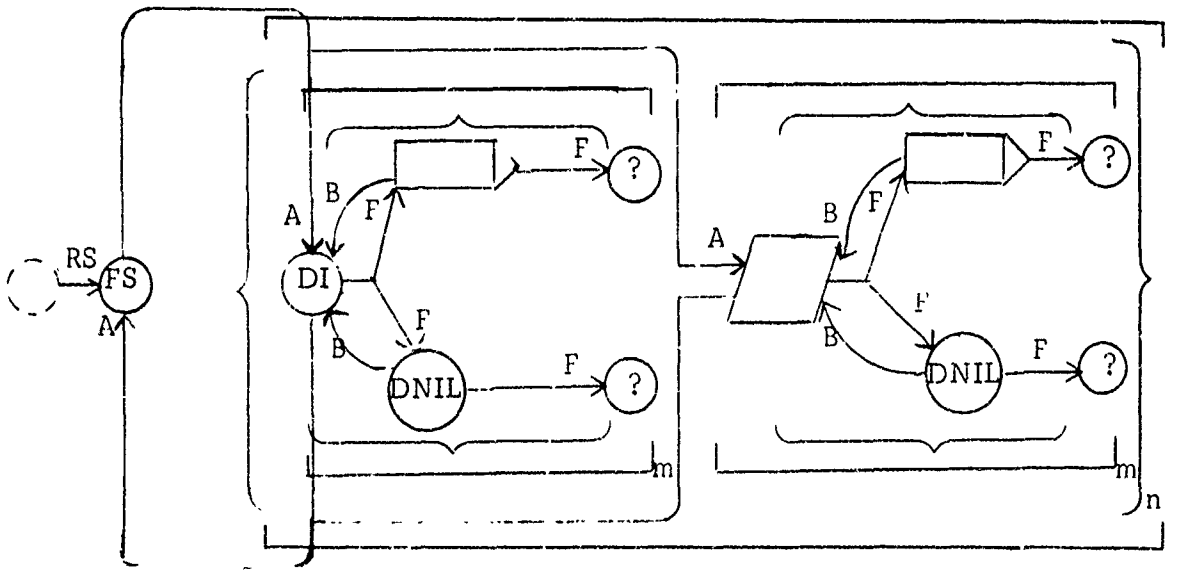
A random access file consists of a number of logical records and a fixed length linear directory. Each directory entry can be accessed by its fixed nonnegative integral index. A directory entry may be null, or may contain the absolute address of a logical record. It may also contain an alphanumeric key for that record.

It is possible for the user program to position the read/write read at an absolute location and thus perform absolute addressing of a random access value. Any program that utilized this would perforce become tied to a given random access device type, and even to a given distribution of files on that volume. We will not attempt to make such programs transferable.

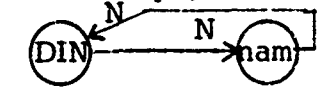
In order to make the user program independent of absolute addresses, SCOPE provides facilities which maintain the directory for a given file and translate user references to record indexes or keys into absolute addresses. We will assume that the user program employs these facilities exclusively and never refers to absolute addresses.

SCOPE random access is a form midway between IBM indexed sequential and partitioned data formats. Like indexed sequential, a random access file is composed of a sequence of logical records, which may have keys. Like a partitioned data set, however, there is an index containing pointers to each record. If the records have keys, these are in the index. Only one directory entry may point to a given record. The directory is fixed in length, and an entry for a particular record does not move about relative to the directory origin.

We may now represent a SCOPE random access file as:



n = 2 implies records with keys, n = 1 implies no keys.

nam stands for 

where nam is any legal character record key

with equivalance 

8.5 SCOPE Abstract Machine

The SCOPE abstract machine for data management, like that of IBM OS/360, has two parts, a buffer handler and a data access machine. These two facilities are in a sense orthogonal, in that all communication between them is restricted to their common access to the circular buffer and the four registers FIRST, IN, OUT, and LIMIT which control the buffer. Thus in order to show the equivalency of two abstract machines for data management we need to show only that their buffer managers and data access machines are separately equivalent.

The abstract machine is defined and controlled for any particular file by its File Environment Table (FET) and by an entry in the File Name Table (FNT). Some of this information, such as absolute disk addresses and device dependent status codes, should never be directly accessed by a program using a SCOPE access method. Whenever possible, these fields will not be mentioned here. Some fields, such as the code and status (CS) field of the FET, contain a mixture of information which should be used and information which should not. We will describe only the parts of such fields which the user program should access, and simply state that the field also contains other data.

8.5.1 File Name Table (FNT)

This is a system table and is protected from user program access. It contains an entry for every file attached to a control point. The following fields are of interest, not because they are accessible to the user but because of what they imply in terms of access method internal structure.

8.5.1.1 Equipment Type

This is set by the system when the file is assigned to a particular device. The user may make a complete or a partial specification of device type and the system will select a device that satisfies the specification. The DT field of the FET is updated from this field every time an I-O command is executed.

8.5.1.2 Last Code and Status

This contains the code and status (CS) field of the FET as it appeared at the completion of the most recent command. This is the field checked by the instruction IF LAST MACRO WAS '<name>', which we invented in order to describe WRITE.

8.5.2 File Environment Table (FET)

The FET is in user memory and is the principal interface between the data management routines and the system. User programs should be very careful about directly accessing the FET, as it contains internal status information for the data management routines. The fields of interest to the user are:

8.5.2.1 Logical File Name (LFN)

8.5.2.2 Code and Status (CS)

Not to be altered by the user.

8.5.2.3 Device Type (DT)

This contains the device type code as copied from the FNT. Device types are so structured that they contain encodements of the following two template name parameters:

8.5.2.3.1 DT.GRP = 1 mass storage device
= 2 tape
= 3 other (telecommunications or unit record)

8.5.2.3.4 DT.DN - device name.

8.5.2.4 R - bit

If R = 1 the file is in random indexed format, and if R = 0 it is in sequential format.

8.5.2.5 Release bit

This causes records to be released after a forward skip or read.

8.5.2.6 UP bit user processing at end of reel

UP = 0 automatic end of reel processing

UP = 1 return to user if end of reel encountered

8.5.2.7 EP bit error processing bit

EP = 1 return to user if error encountered

EP = 0 kill job if error encountered

8.5.2.8 Disposition Code (DC) - Specifies disposition of file after CLOSE

8.5.2.9 PRUSZ physical record unit size

8.5.2.10 RBSZ record block size. Number of PRU's in a physical record. Not useful to user program.

8.5.2.11 FIRST, IN, OUT, LIMIT - registers for circular buffer

8.5.2.11.1 WSA - registers defining a working storage area

8.5.2.12 (S or L tapes only) UBC - unused bit count. This is used to indicate the number of garbage bits in the low order part of the last data word in the circular buffer.

8.5.2.13 (S or L tapes only) MLRS - maximum logical record size.

8.5.2.14 (random files only) record request/return - not to be accessed by user.

8.5.3 SCOPE Buffer

The buffer is defined by four registers: FIRST, IN, OUT, and LIMIT. These are discussed in section 8.1.7 of this paper. We will define here five primitive operations on these registers. Let m = (LAST-FIRST).

8.5.3.1 CLEARBUFF
i) sets OUT = IN

8.5.3.2 ON BUFFULL GOTO S
where S is a statement label
i) branches to S if ((OUT-IN) mod m) = 1

8.5.3.3 ON BUFCLR GOTO S
i) branches to S if OUT=IN

8.5.3.4 READN
i) put contents of current node into @ IN
ii) IN ← FIRST + ((IN - FIRST + 1) mod m)

8.5.3.5 WRITEN
i) put contents of @ OUT into current node
ii) OUT ← FIRST + ((OUT - FIRST + 1) mod m)

8.5.3.6 ON PRUNFIT GOTO S
i) branches to S if ((OUT-IN) mod m) ≤ PRUSZ

8.5.3.7 ON NPRUBUF, GOTO S
i) branches to S if ((IN-TO) mod m) < PRUSZ

8.5.4 RDPRU Read One PRU

Because of its importance, we will define a separate macro for the process of reading one PRU into the circular buffer.

```
RDPRU    MACRO      ¢ℓ, ¢TRLT, ¢TREQ, ¢TRGT
$S1      READN
         MOVEH      F
         ONNODE     (D,ELSE) GOTO (¢S1,NEXT)
         ONNODE     (PND,ELSE) GOTO (¢S3, NEXT)
         ONNODE     (LØ, ELSE) GOTO (¢TRLT1,NEXT)
         ONNODE     (L1,ELSE) GOTO (¢TRLT1,NEXT)
            .
            .
            .
         ONNODE     (L(¢ℓ-1),ELSE) GOTO (¢TRLT1,NEXT)
         ONNODE     (L¢ℓ,ELSE) GOTO (¢TREQ1,NEXT)
         ONNODE     (L(¢ℓ + 1),ELSE) GOTO (¢TRGT1,NEXT)
            .
            .
            .
         ONNODE     (L15,ELSE) GOTO (¢TRGT1,ERROR)
¢TRLT1   MOVEH      F
         GOTO       ¢TRLT
¢TREQ1   MOVEH      F
         GOTO       ¢TREQ
¢TRGT1   MOVEH      F
         GOTO       ¢TRGT
¢S3      NOOP
RDPRU    MEND
```

RDPRU works quite unambiguously whenever the current node is the first type D node of a PRU and it is known that the circular buffer has space left for at least one PRU. We will use RDPRU only under these conditions. It should be noted, however, that PRU's have physical significance as a

hardware unit of transaction and that normally all of a file will be in the form of PRU's, including tape labels and random access directories. Thus one might object that our restrictions on the use of RDPRU are too severe. It should be noted, however, that SCOPE does not explicitly commit itself to a particular format for the PRU encodement of its file labels and indexes. If we fixed into our model the particular encodement that now obtains we would in effect be binding SCOPE to that encodement. This overspecification would make SCOPE harder to improve and less flexible.

8.6 SCOPE Data Access Macros Useable On Sequential Files

8.6.1 READ

When READ is called the current node must be either the first node of a PRU or the end of the file.

```
READ    MACRO
¢S2     ON PRUNFIT GOTO ¢S1
        RDPRU Ø, ¢S1, ¢S1, ¢S1
        GOTO ¢S2
¢S1     NOOP
READ    MEND
```

8.6.2 READSKP ℓ

READSKP functions like READ except that if the circular buffer is filled before a record end is reached or if a record end is reached which is less than ℓ the virtual head is moved forward either until just after the first record end with level equal to or greater than ℓ or until the end of file is reached, whichever happens first. It should be noted that since the end of file indicator is a zero-length PRU with the maximum possible level, the two cases amount to the same thing.

```
READSKP   MACRO                  ¢LEV
¢S2       ON PRUNFIT GOTO        ¢S1
          RDPRU                  ¢LEV, ¢S1, ¢S3, ¢S3
          GOTO                   ¢S2
¢S1       SKPRU                  ¢LEV, ¢S1, ¢S3, ¢S3
          GOTO                   ¢S1
¢S3       NOOP
READSKP   MEND
```

where SKPRU is exactly like RDPRU except that READN is replaced by NOOP, and where ¢LEV is an integer from zero to 15 inclusive.

8.6.3 WRITE

This causes full PRU's to be written out from the circular buffer until the buffer no longer contains enough data for a full PRU.

```
WRITE     MACRO
¢S2       ON NPRUBUF GOTO ¢AROUND
          XFORM      FP
¢S1       MOVEH      F
          ON NODE (D,ELSE) GOTO (NEXT,¢S2)
          WRITEN
          GOTO ¢S1
¢AROUND   MOVEH      F
WRITE     MEND
```

8.6.4 WRITER

This is like WRITE, except that when the circular buffer contains less than PRUSZ number of words a truncated or zero-length PRU is written out with appropriate level.

```
WRITER    MACRO       ¢ℓ
¢S2       ON          NPRUBUF GOTO ¢AROUND
          XFORM       FP
¢S1       MOVEH       F
          ON NODE     (D,ELSE) GOTO (NEXT, ¢S2)
          WRITEN
          GOTO        ¢S1
¢AROUND   XFMTPL      ¢ℓ
¢S3       MOVEH       F
          ONNODE      (D,ELSE) GOTO (NEXT,¢DUN)
          WRITEN
          GOTO        ¢S3
¢DUN      MOVEH       F
WRITER    MEND
```

8.6.5 WRITEF

WRITEF operates differently depending on whether or not the SCOPE I-O command most recently performed was WRITE. This requires an addition to the SCOPE abstract machine. We need a register, LSTMKRO, which contains the name of the last SCOPE macro issued. We will test this register by the instruction

IF LAST MACRO WAS '(name)', GOTO

where <name> is the last name of the last macro issued.

```
WRITEF    MACRO
          ON BUFCLR GOTO ¢S1
          WRITER ø'
          GOTO ¢WEOF
¢S1       IF LAST MACRO WAS 'WRITE', GOTO ¢S2
          GOTO ¢WEOF
¢S2       XFORM TPLQ(ø,ø)
¢WEOF     XFORM TPLQ(15,0)
WRITEF    MEND
```

8.6.6 SKIPF n,ℓ

First, define a simpler form, SKIP1F ℓ, as follows

```
SKIP1F    MACRO     ¢ℓ
¢S1       SKPRU     ¢ℓ, ¢S1, ¢S2, ¢S2
          GOTO      ¢S1
¢S2       NOOP
SKIP1F    MEND
```

SKIPF is then

```
SKIPF     MACRO     ¢n,¢ℓ
          SKIP1F    ¢ℓ
¢n times    .
            .
            .
          SKIP1F    ¢ℓ
SKIPF     MEND
```

8.6.7 SKIPB - Skip Backwards

Needless to say, this command is legal only for doubly sequential data structures. To define SKIPB, we will use a macro

SKPRUB n,SLT,SEQ,SGT

This uses MOVEH B to space backward until a type PND, BRL or BF node is reached. Control will exit from SKPRUB as follows:

- Next sequential instruction if no level node was found
- SLT if a level node Li was found and i < n
- SEQ if Li was found and i = n
- SGT if Li was found and i > n

The next intermediate stage is SKIP1B

```
SKIP1B      MACRO       ¢ℓ
¢S1         SKPRUB      ¢ℓ, ¢S1, ¢S2, ¢S2
            GOTO        ¢S1
¢S2         SKPRU       Ø, ¢S3, ¢S3, ¢S3
¢S3         NOOP
SKIP1B      MEND
```

SKIP1B is then

```
SKIPB       MACRO       ¢n, ¢ℓ
            SKIP1B      ¢ℓ
¢n times    .
            .
            .
            SKIP1B      ¢ℓ
SKIPB       MEND
```

8.6.8 BKSP

This causes the read to be backspaced one logical record. It is equivalent to

```
            SKIPB       1,Ø
```

8.6.9 BKSPRU

```
BKSPRU      MACRO       ¢n
¢S1         SKPRUB      0, ¢S2, ¢S2, ¢S2
¢S2         SKPRUB      0, ¢S3, ¢S3, ¢S3
 :           :                  :
¢Sn         SKPRUB      0, ¢S(n + 1), ¢S(n + 1), ¢S(n + 1)
¢S(n + 1)   NOOP
BKSPRU      MEND
```

In other words, space backwards n PRU's.

8.6.10 REWIND, UNLOAD

It is not at all clear how REWIND and UNLOAD differ because it is not at all clear just how a rewind affects a file. We will for the moment ingnore UNLOAD, saying that it is similar to REWIND. REWIND is as follows:

```
REWIND    MACRO
          MOVEH       R
          MOVEH       F
REWIND    MEND
```

8.6.11 RPHR, WPHR

These read or write one PRU of 512 words on SCOPE standard magnetic tape only. RPHR clears the circular buffer before reading a PRU, WPHR clears the buffer after writing a PRU. READ and WRITE could be used to accomplish practically the same result, and we will not discuss RPHR or WPHR further.

8.6.12 READN

READN is a nonstop read and can be used only on type S or L tapes. Because of its limited applicability we will not discuss it here. It is worth noting, however, that SCOPE adds to the beginning of the record in the circular buffer a header giving the record length. SCOPE, of course, must wait until it reaches the end of the record to generate this information.

8.6.13 READIN

READIN has three forms, differing in the parameter supplied. The no-parameter form can be used on a sequential file. The other two forms are restricted to indexed files. READIN transfers data from the circular buffer to a secondary buffer called the working storage area. This is conceptually not a data access function but rather a program level subroutine which happens to call on the data management routine. We shall therefore discuss only how READIN calls on the data management routine and not attempt to define what a working storage area is.

READIN attempts to fill the working storage area from the circular buffer. It will call READ if in the process of doing this it finds the circular buffer empty.

8.6.14 WRITEN

This is similar to READN. It is for S and L magnetic tapes only. Records placed in the circular buffer will be written out. A header must precede each record indicating that record's length.

8.6.15 WRITOUT

This is similar to READIN. The no-parameter form is supposedly usable on sequential files, but the description is contradictory: a WRITOUT may be issued only if there is a "current record"; a "current record" exists only if a WRITOUT has been issued. There is throughout the SCOPE manual great ambiguity about what the "current record" is. As READIN and WRITOUT are of marginal usefulness for sequential files anyway, we won't describe them further in this section.

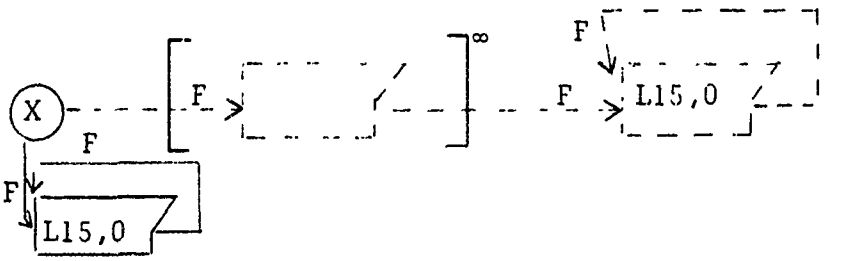
8.6.16 REWRITE, REWRITER*ℓ*, REWRITEF

These may be used only on mass storage files, either sequential or indexed. They cause PRU's to be rewritten starting from the current head position. Any information contained in the old PRU's is lost, including end of record and end of file information. For instance, if a full PRU of a multi-PRU record is rewritten as a truncated PRU, the original record will have been split into two records. REWRITE will also blithely write over indexes and labels. SCOPE warns against this and promises unpredictable results. As we discussed in the section on RDPRU, we will not guarantee that our unpredictable results match SCOPE's unpredictable results when restrictions are violated.

In order to define REWRITE we need some new transformations:

```
XFORM     RFP
XFORM     RTP (ℓ,q)  0≤ℓ≤15,  0≤q≤PRUSZ
XFORM     REF
```

The current head position must be at the first node of a PRU. This PRU will be replaced by a full PRU if XFORM RFP is called and by a truncated or zero length PRU of length ℓ and level q is XFORM RTP (ℓ,q) is called. XFORM REF replaces the current PRU by an end of file PRU. Because of the special way end of files are handled, REF≠RTP(15,0). Instead, all of the file from the current node to the end is deleted and a new end of file PRU appended:



The PRU's deleted still exist on the storage volume. Since they now belong to no file, they are not legally accessible however.

REWRITE is just like WRITE except that XFORM FP is replaced by XFORM RFP:

```
REWRITE   MACRO
¢S2       ON          NPRUBUF GOTO ¢AROUND
          XFORM       RFP
¢S1       MOVEH       F
          ONNODE      (D, ELSE) GOTO (NEXT,¢S2)
          WRITEN
          GOTO        ¢S1
¢AROUND   NOOP
WRITE     MEND
```

For REWRITER, we need a new version of XFMTPL, namely XFMRTPL ℓ. This verifies that the data, if any, in the circular buffer will fit in a truncated PRU then executes

XFORM RTP(ℓ,q)

where q = ((IN-OUT) mod m)

REWRITER is then:

```
REWRITER    MACRO       ¢ℓ
¢S2         ON          NPRUBUF GOTO ¢AROUND
            XFORM       RFP
¢S1         MOVEH       F
            ONNODE      (D,ELSE) GOTO (NEXT,¢S2)
            WRITEN
            GOTO        ¢S1
¢AROUND     XFMRTPL     ¢ℓ
¢S3         MOVEH       F
            ONNODE      (D,ELSE) GOTO (NEXT,¢DUN)
            WRITEN
            GOTO        ¢S3
¢DUN        MOVEH       F
REWRITER    MEND
```

REWRITEF is simply

```
REWRITEF    MACRO
            XFORM       REF
REWRITEF    MEND
```

8.6.17 WRITIN

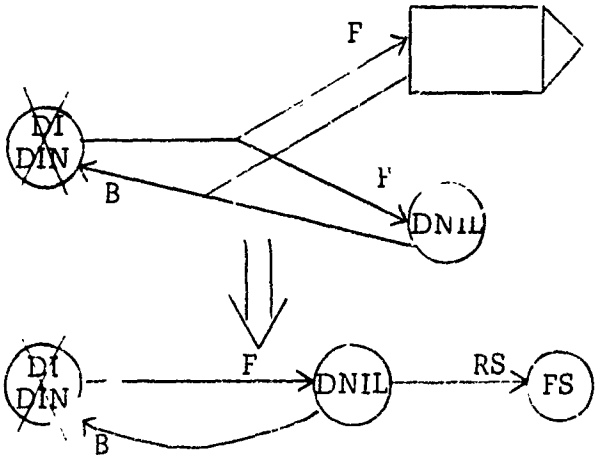
This is similar to WRITOUT except that where WRITOUT would use a WRITE, WRITIN would use a REWRITE.

8.7 Macros and Transforms Usable on Random Access Files

8.7.1 XFORM DELREC

This transform is used to delete a record in a random access file. The covering template is TRA(n,m) that is, the general random access template with a definite choice made between named (n = 2) or numbered (n = 1), records and with a definite number (m) of directory entries. The current node must be of type DI or DIN.

8.7.1.1 $DELREC_2$



Which means that the current state, which must be pointing either to a record or to a DNIL node, is made to point to a DNIL node.

8.7.1.2 $DELREC_1$, $DELREC_3$

Most of the file structure remains unchanged. The DNIL node or record which is the F successor of the current node is deleted, and all arcs exiting from that node or record are deleted. In its place a new DNIL node is substituted. This node is the F successor of the current node. The current node is made the B successor of the DNIL node, and the FS node is made the RS successor of the DNIL node. No other nodes or arcs are affected.

8.7.2 READ, READSKP, READNS, WRITE, WRITER, WRITEF

These commands are designed for sequential files. Because SCOPE random indexed records are the same as sequential records, these commands may be used to continue reading or writing a record once it has been located in the index. They may also be (mis-) used to read or write past the end of the current record, with unpredictable results. This is certainly to be discouraged in a transferable system.

8.7.3 READIN, READIN /name/, READIN m

As discussed in the section on sequential files, READIN is more a user program subroutine than a data management macro and we will discuss only how it uses more basic data access macros. READIN moves data from the circular buffer to a secondary buffer called the working storage area and calls READ if the circular buffer becomes empty before this transfer is complete. If the parameter /name/ or the integer is specified the current head position will be moved to the beginning of the corresponding record, the circular buffer will be emptied, and a READ will be issued. We will give macros here which will search the index and position the head.

DEXNAM assumes a keyed index format. It uses the special command

```
COMPNAM          nam, ST, SF
```

where name is a 7 character name, and ST and SF are statement labels. The current node must be of type DIN. If nam matches the current index key, control will pass to statement ST, otherwise control will pass to SF.

```
DEXNAM      MACRO       ¢NAM
            MOVEH       RS
¢S1         MOVEH       A
            ONNODE      (DIN) GOTO (NEXT)
            COMPNAM     ¢NAM, NEXT, ¢S1
DEXNAM      MEND
```

Note that the ONNODE instruction will force an error if the name is not found.

DEX m for m a nonnegative integer is not a single macro but rather a set of macros: DEX0, DEX1, ... where DEXM m stands for DEXm.

There are two basic forms:

```
for m = 0       DEX0        MACRO
                            MOVEH     A
                DEX0        MEND

and for m > 0   DEXm        MACRO
                            MOVEH     RS
                            MOVEH     A
                            .
                m times     .
                            .
                            MOVEH     A
                DEXm        MEND
```

Now that DEXNAM and DEXM have been defined, we can say that READIN /name/ calls DEXNAM /name/ and READIN m calls DEXM m before issuing a READ.

8.7.4 WRITOUT, WRITOUT /name/, WRITOUT m

This is the companion to READIN. WRITOUT will clear the circular buffer, transfer data from the working storage area to the circular buffer, and issue a WRITE. WRITOUT m will call DEXM m before issuing the WRITE. WRITOUT /name/, however, apparently does not call DEXNAM /name/. In fact it is quite unclear from the manual just what it does.

8.7.5 REWRITE, REWRITERℓ, REWRITEF

Just as with READ, READSKP, etc., these three commands may be used at any point in a random indexed file. If a programmer makes direct use of these commands, however, he must make certain that he does not damage the indexed structure. WRITIN is a user level subroutine that uses REWRITE(R,F) in a manner consistent with the indexed file structure. For transferable programs, we would not recommend direct use of REWRITE on random indexed files.

8.7.6 WRITIN, WRITIN /name/, WRITIN m

WRITIN is like WRITOUT, but with two differences. First WRITIN calls REWRITE whenever WRITOUT would have called WRITE. Second, WRITIN /name/ does call DEXNAM /name/ before issuing the REWRITE.

8.8 File Structure Templates

Earlier we defined templates separately for each SCOPE file structure. It will be instructive to fit these templates into a template name grammer. We can then describe the process of specifying file structure as a series of REFINE TEMPLATE commands.

The most general template name is SFILE. This contains all legal SCOPE file structures. It has only one explicit parameter, R, which can take the values 0 or 1. It is stored in the r bit of the FET. The two possible refinements are then:

(SFILE, R = 0) = SEQSF
sequential SCOPE file

(SFILE, R = 1) = RISF
random indexed SCOPE file

Here we have defined the alias names SEQSF and RISF, which may replace (SFILE, R = 0) and (SFILE, R = 1), respectively, any place they occur.

RISF has one explicit parameter, KEY, which can take on the integer values 1 or 2 corresponding to records with no keys and records with keys, respectively. We will again develop aliases:

(RISF, KEY = 1) = RISFNK

(RISF, KEY = 2) = RISFK

Both RISFK and RISFNK have the same explicit parameters: PRUSZ and LI. Both take positive integer values. PRUSZ is a field in the FET and specifies the number of characters in a PRU. LI is the number of entries in the index.

SEQSF has one explicit parameter, DT.GRP, which can take on the values 1 2, or 3. It groups file structures by device type: 1 for mass storage, 2 for tape, and 3 for other types (mostly telecommunications and unit record). (SEQSF,DT.GRP = 3) is not supported by SCOPE data management and will not be considered further here. (SEQSF,DT.GRP = 2) has one explicit parameter, DT.DN. This may take on the values S, L, and STD, with the following aliases and meanings:

((SEQSF,DT.GRP = 2), DT.DN = STD) = STDTAPE
SCOPE standard 1/2" magnetic tape

((SEQSF,DT.GRP = 2), DT.DN = S) = STAPE
Stranger tape

((SEQSF,DT.GRP = 2), DT.DN = L) = LTAPE
Long record stranger tape

The device type, or DT, field of the FET contains an encodement of DT.GRP and DT.DN.

(SEQSF,DT.GRP = 1) = MSEQ is a sequential file on a random access mass storage device.

MSEQ and STDTAPE each have one explicit parameter, PRUSZ. STAPE and LTAPE each have one parameter, MLRS. Both PRUSZ and MLRS are fields of the FET.

9. HONEYWELL (GE) 600 GEFRC (GEneral File and Record Control)

9.1 Introduction

The Honeywell GEneral File and Record Control program (GEFRC) operates on any Honeywell 600/6000 series machine in cooperation with the GEneral Comprehensive Operating Supervisor (GECOS). GEFRC provides input output servicing in a simpler form than IBM OS/360 or CDC SCOPE. Although simple in form, GEFRC provides complete input/output service for all the common peripheral devices such as unit record equipment (card, printer, paper tape), magnetic tape subsystems, and disk and drum subsystems as well as remote devices such as teletypes and batch remote stations.

Since the format of a file varies greatly with the type of device to which it is assigned, a Standard System Format (SSF) has been designed. For certain devices, the restrictions of the SSF are relaxed. Files which use the SSF can be moved from any device to any other device within the system without changing the user's program.

In this chapter we shall discuss the GEFRC file structure, the form of various control blocks, GEFRC buffering, and the detailed i/o instructions themselves: logical record processing, device positioning, physical record processing, input/output editing, and file preparation.

9.2 File Structure - Standard System Format

A Standard System Format (SSF) file is a sequence of zero or more blocks (physical records) preceded by a header record and followed by a trailer record. Under certain circumstances a file may be unlabeled; i.e., not have header and trailer records. This is true for unit record equipment and may be true for magnetic tape. An SSF block is variable in size, with a maximum of 320 words. The first word contains block serial number and block size fields. Again, certain devices support fixed size blocks which do not have the block serial number field. Their size is determined from the maximum block size field of the fcb (see 9.3 below). An SSF record may be in one of three forms: variable, fixed, or mixed. In all three cases, a record is a sequence of n

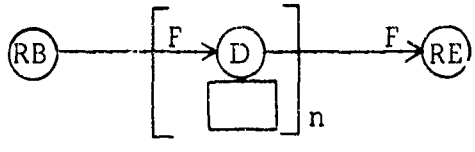
words; the only difference in the forms is the method used to determine n. Variable length records are in fact of length n + 1. The first word is the record size control word (RSCW), which contains certain control information (file mark if size is zero, logical record type for media conversion, report code) and the size, n, of the record. Fixed length records are of size n where n is determined from the record size field of the fcb. Mixed length records are of size n where n is determined by invoking a user provided routine to look at the record to determine its size. The address of this routine is contained in the fcb. It may be observed that all three formats are really special cases of mixed. In the case of fixed records, a system provided size routine is called which determines the size from the fcb; in the case of variable, a system provided size routine is called which determines the size from the first word (RSCW) of the record. In all three cases the record is in a user provided buffer rather than on the device itslef.

In the following sections we give the templates and macros for standard system format records, blocks and files.

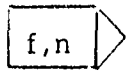
9.2.1 Records

A record may be fixed length, mixed length, or variable length.

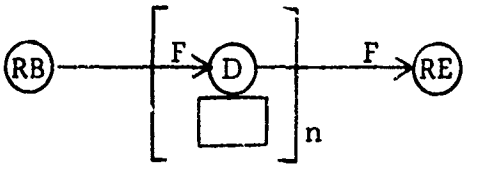
9.2.1.1 Fixed Length Records



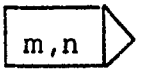
where n is the same for all records in the data set. We shall denote this template by



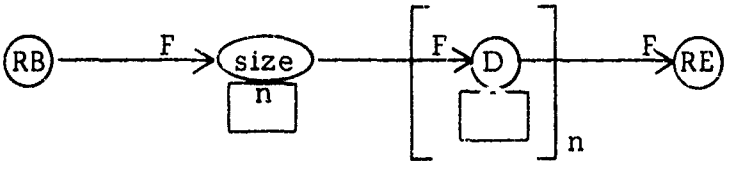
9.2.1.2 Mixed Length Records



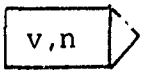
where n is determined by invoking a user provided routine immediately prior to reading the record. We shall denote this template by



9.2.1.3 Variable Length Records



where n is determined by reading the value in the size node. Note that a variable length record could be read as a mixed length record of size n + 1 if the user implemented a user routine to extract the number stored in the first word of the record. Similarly, fixed length records could be read as mixed length records. We shall denote the variable length template by

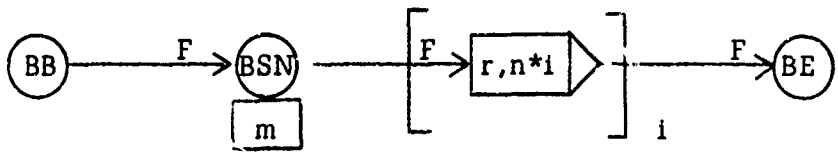


9.2.2 Blocks

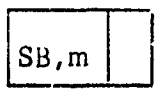
A block is a linear sequence of records of one of the three types: fixed, mixed, or variable. A block may or may not have block sequence numbers.

9.2.2.1 Standard Block

A standard block contains a block size node and may be represented as

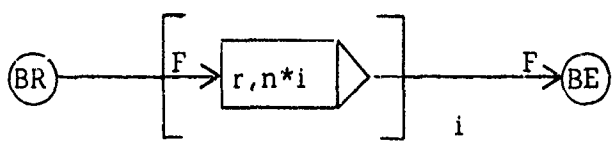


where m is the length of the block in words and r is f, m, or v. The block contains i records of type r. For an r of f or m, $\sum_i n*i = m$. For an r of v, $\sum_i (n+1)*i = m$. For standard system format, m must be less than or equal to 320. A GEFRC limit is $m \leq 4094$. We shall denote the standard block template by

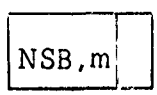


9.2.2.2 Non-Standard Block

A non-standard block does not contain a block size node. The block size information must be determined from the fcb, and is fixed in size for the file.



This template may be denoted by

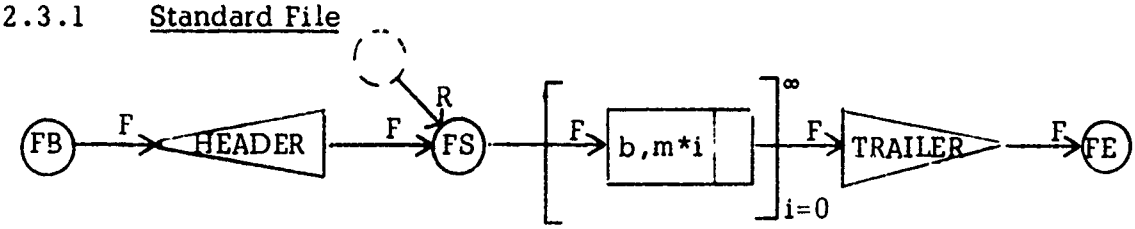


where m is the record size and r the record type, as for standard blocks.

9.2.3 Files

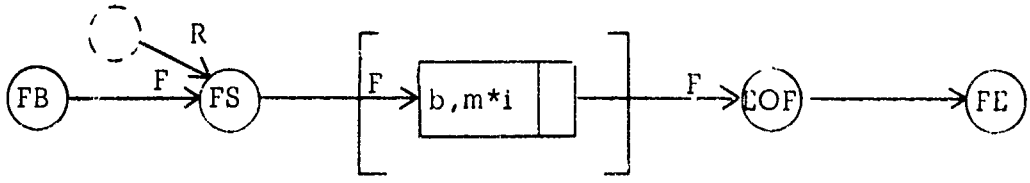
A file is a linear sequence of either standard or nonstandard blocks. A file may or may not have header or trailer records.

9.2.3.1 Standard File



where b is either SB or NSB. The structures of HEADER and TRAILER are given below. The template may be denoted by

9.2.3.2 Non-Standard File



where b is either SB or NSB, and EOF is physical (device) end of file. The template may be denoted by

9.2.4 Header and Trailer Labels

GEFRC includes a complete facility for processing standard labels and for performing associated unit switching at the end of magnetic tape reels. The procedures included are specifically designed for the standard case and as such will not perform label functions on nonstandard labels. We have not attempted to model multi-reel files or multi-file reels and will give only verbal explanations for the appropriate routines.

Header and trailer labels are standard, 14 word blocks.

9.2.4.1 Header

word number	format	description
1-2	GEØØ600ØBTLØ	label identifier
3	xxxxxx	installation identification
4	Øxxxxx	tape reel serial number
5	Øxxxxx	file serial number
6	ØØxxxx	reel sequence number
7	Øyyddd	creation date
8	ØØØxxx	retention days
9-10	xxxxxx	file name
11-14	(arbitrary)	not used - available for user program

9.2.4.2 Trailer

word number	format	description
1	ØEORØØ	end-of-reel
	or	
	ØEOFØØ	end-of-file
2	xxxxxx	block count
3-14	(arbitrary)	not used

9.3 File Control Block

Like most file systems, GEFRC uses a file control block (FCB). The FCB is a fixed format block of information about the file. This information comes from various sources: the programmer when the FCB is set up, the file control cards via the operating system (GECOS), the low level input output system (IOS), and GEFRC itself. GEFRC uses another storage block, the file designator word (FDW) to contain information about open and close options. For our purposes we will ignore the existence of the FDW and assume that all information about the file not contained within the file itself is contained with the FCB.

The existence of the FCB in a form accessible to a programmer causes serious problems in the accurate description of the system. Since fields of the FCB are accessible and can be modified by a programmer, questions as to the legality of such access occur. For example: consider a file with a MAX-BLOCK of 400, a RECORD-FORM of FIXED and a RECORD-SIZE of 80. A GET command returns a logical record of 80 characters. What happens if, on the fly, the programmer changes the RECORD-SIZE field to 100? Three possibilities come to mind: 1) the change is ignored, 2) GEFRC aborts with a more or less cryptic error message, or 3) the logical record size changes from 80 to 100. This type of question occurs because it is not clear from a manual just when certain information is conveyed from the program (FCB) to the file system. If the RECORD-SIZE field is looked at only at open time, then a change will be ignored; if it is looked at with each GET, then it essentially is an argument to the GET and the change to the RECORD-SIZE should take effect.

We will attempt in our model of the file system to replace the FCB completely with the template and appropriate refinements. If we succeed, we shall have a model of the file system which is hopefully easy to follow and unambiguous and which may lead to quantitative analysis in the future. If we fail, the failure should shed light on a) defects in the model, and b) warts in the GEFRC file system. The goal is similar to the goal of those who build models of programming languages: we are attempting to build an abstract syntax for file systems and an abstract machine to perform operations on the file system.

For reference, the more important fields of the FCB (and FDW) are given below.

9.3.1 File Control Block

NAME	symbolic name of file control block
FILE CODE	symbolic name of file to link with control card
BUFFER 1	symbolic name of buffer, if not present implies physical input/output only
BUFFER 2	symbolic name of buffer, if present implies double buffering
MAX BLOCK	size of largest block, must be ≤4095, default is 320
RECORD FORM	variable, fixed, or mixed, default is variable
RECORD SIZE	decimal number if fixed record form, symbolic name of procedure if mixed record form
BLOCK SERIAL NUMBERS	included in file or not
ERROR	symbolic name of user error routine
LABELS	standard labels present or not (tape only)
MODE	binary, bcd, or mixed (card input only)
DENSITY	low or high for magtape
MULTIFILE	for tape only, more than one file for this reel
RETENTION	number of days
PREHEADER	symbolic name of user routine
POST HEADER	" " " " "
PRE TRAILER	" " " " "
POST TRAILER	" " " " "
FILE NAME	for header checking (input) or putting into header (output)

9.3.2 File Designator Word

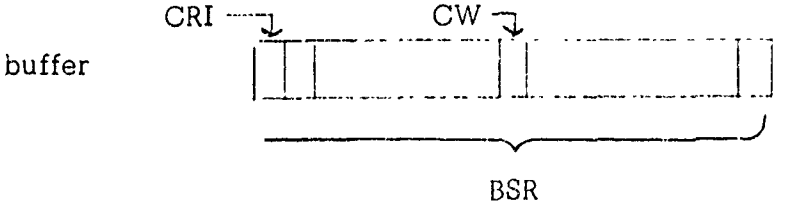
FCB	file control block name
IO	input or output
OPEN	rewind on open or not
CLOSE	rewind on close or not

PRIME	for buffered input file, whether or not to fill buffer at open
SIZE	for buffered output file, programmer will call putsz (0) on close
REQ	abort if file not present or not
FILE	position to file n on multifile tape

9.4 Buffering

GEFRC requires that the user take responsibility for buffering. The user must decide whether or not he wants buffering and if he does (necessary for logical record processing) whether he wants single or double buffering. It is also the user's responsibility to set aside space for any buffers and buffer control words which may be needed. With logical record processing, a block, or physical record, is the unit of transaction with the device. It is a single block which resides in a buffer. Logical records are treated by manipulating the current record index (the address of the logical record) and the record size, both of which are fields within the fcb.

We have found that it is not necessary to have all this machinery to explain, logically, what is happening. We give here our model of the GEFRC buffering scheme. A buffer is determined by three items, the CRI or current record index which is the address of the buffer, the BSR or buffer size register which is the number of words in the buffer, and the CW or current word which is that buffer word currently of interest. The words in the buffer are numbered starting from zero. Thus the maximum legal value of CW is BSR-1. The address of the word indicated by CW is CRI + CW.



There are various commands which operate on buffers:

GVBUF - frees the buffer pointed to by CRI

GBUF - allocate a buffer of size BSR. Set CRI to point to the buffer. Set CW to zero.

READN - put contents of current node into word pointed at by CW. CW ← CW + 1

WRITEN - put contents of word pointed at by CW into current node. CW ← CW + 1.

To handle the GEFRC block structure, two commands and a predicate are needed:

BLOCKOUT - does XFORM to create a block in the file, and sets block length to zero.

RECORDOUT - does XFORM to create a record in the file, and does block length ← block length + BSR.

BUFFERNOTFULL - if block length + BSR ≤ MAXBLOCK then true else false.

9.5 Logical Record Processing

The user may read logical records from input files and write logical records to output files. The GEFRC routines which perform these logical reads and writes also accomplish the necessary blocking, deblocking, and the physical record reading and writing in accordance with information in the file control block. As an option, the logical read and write requests may cause the logical records to be physically moved between the buffers and specified working storage locations. After a file has been processed, it must be closed. When a file is closed, the buffers are emptied and label processing and repositioning occur as specified by the calling sequence and the file control block.

9.5.1 GET

The GET macro obtains the next logical input record from a designated input file. The calling sequence is

CALL GET (fcb, eof [,stor])

where fcb is the name of a file control block.
eof is the name of a user's end-of-file routine
stor is the name of a working storage area into which the record is to be copied (optional).

Following a call to GET, the current record index points to the record in the buffer and the record size field contains the number of words in the record. Logical record processing assumes that an RB node is the current node. Each macro must assure that, at its completion, this is true. The GET macro is as follows:

```
GET         MACRO (fcb, eof [ ,stor])
            ONNODE (RB,ELSE) GOTO (NEXT,eof)
            GVBUF
            MOVEH   F
            BSR ← record size          (Note 1)
            GBUF
            FOR i ← 1,...,BSR DO [[ MOVEH F; READN]]
            MOVEH F; MOVEH F
            ONNODE (RB,BE) GOTO (¢DONE,NEXT)
¢L          MOVEH   F
            ONNODE (BB, TRAILER, EOF) GOTO (NEXT,¢DONE,¢DONE)
            MOVEH   F
            ONNODE (RB,BE) GOTO (¢DONE,¢L)
¢DONE       [ copy buffer to stor ]        (Note 2)
GET         MEND
```

Note 1: The determination of the record size depends on the record type. For fixed length records, the length was placed into BSR at OPEN. For mixed length records, a user routine (specified in the fcb) is invoked:

BSR ← user_routine

For variable length records, the value in the size node is read:

BSR ← cont (size_node)

Note 2: If user storage location is specified, contents of buffer is copied to that location.

9.5.2 GETBK

The macro GETBK obtains the first logical record in the next physical record from a designated input file. The calling sequence is

CALL GETBK (fcb, eof[,stor]).

The GETBK macro performs the same functions as the GET macro except that all logical records remaining in the last accessed physical record are ignored. The GETBK macro is as follows:

```
GETBK       MACRO   (fcb,eof[ ,stor])
¢L          MOVEH   F
            ONNODE (BB,TRAILER,EOF,ELSE) GOTO (NEXT,eof,eof,¢L)
            MOVEH   F
            ONNODE (RB,BE,BSN) GOTO (¢LL,¢L,NEXT)
            MOVEH   F
¢LL         GET (fcb,eof[ ,stor])
GETBK       MEND
```

9.5.3 PUT

The PUT macro allocates space within a buffer for the designated output file for inserting the next logical record of that file and, if desired, moves that logical record to the allocated area. The calling sequence is

CALL PUT (fcb[stor]).

Following a call to PUT, the current record index points to the logical record and the record size field indicates its size. The record size field must have been set prior to the call to PUT. Note that PUT does not transmit the record; rather it allocates a buffer space for the record. The record is not transmitted until the next call to PUT. Thus a programmer may modify a record in the

output buffer (see PUTSZ, 9.5.6 below). The PUT macro is as follows:

```
PUT         MACRO     (fcb[,stor]
            RECORD OUT
            MOVEH  F
            FOR i← 1,...,BSR DO [WRITEN;MOVEH  F]
            GVBUF
            BSR ← record size
            ON BUFFERNOTFULL GOTO ¢X
            BLOCKOUT
¢X          GBUF
            [copy stor to logical record]
PUT         MEND
```

9.5.4 PUTBK

The PUTBK macro allocates space at the beginning of a buffer for the designated output file for inserting the next logical record of that file and, if desired, moves that logical record to the allocated area. The calling sequence is:

```
CALL PUTBK (fcb [,stor])
```

The call to PUTBK performs as a call to PUT except that the logical record will be first in a new physical record. This implies that the physical record which has been under construction in the buffer may be shorter than the usual physical record for this file. The PUTBK macro, which follows, uses a call to PUT with a large record size to force termination of the current physical record, followed by a call to PUTSZ (9.5.6) to reposition at the beginning of the block, followed by a call to PUT to actually allocate the record:

```
PUTBK       MACRO (fcb[,stor])
            t ← record size
            record size ← MAXBLOCK
            PUT (fcb)
            PUTSZ (fcb,0)
            record size ← t
            PUT (fcb,stor])
PUTBK       MEND
```

9.5.5 COPY

The COPY macro moves the last accessed logical input record from the designated input file to the next available position in the designated output file. The calling sequence is

CALL COPY (fcb-out, fcb-in)

The CALL COPY command performs the same function as CALL PUT except that the current record index of the input file is used in place of the usual working storage location (stor in CALL PUT). The size of the record is determined by the record size field of the output file control block. The input file control block (fcb-in) is not modified or checked in any manner. See description of PUT for return and exception condition information. The COPY macro is as follows:

```
COPY        MACRO    (fcb-out, fcb-in)
            PUT      (fcb-out,current record index fcb-in)
COPY        MEND
```

9.5.6 PUTSZ

The PUTSZ macro is used to update the file control block of the designated output file to reflect the true size of the last logical record placed in that file. The calling sequence is:

CALL PUTSZ (fcb, size).

This macro is generally used in the case where an output record of unknown length is to be constructed in the buffer. Either CALL PUT or CALL PUTBK is issued with the record size field of the file control block set to some maximum record size value. Space for this maximum size record is thus reserved. After the record has been constructed and its actual length determined, the CALL PUTSZ command is issued to update the file control block with the appropriate pointers. The PUTSZ macro is as follows:

```
PUTSZ       MACRO    (fcb,size)
            BSR ← size
PUTSZ       MEND
```

9.5.7 RELSE

The RELSE macro causes the next referenced logical record of the designated file to be the first logical record of the next physical record. The calling sequence is

CALL RELSE (fcb)

If the file designated by fcb is an input file, then any logical records remaining in the current physical record will be ignored. The next logical record request will obtain the first logical record in the next physical record. If the file designated by fcb is an output file, then the physical record currently under construction will be written. This physical record may be shorter than the usual record created for this file. The next logical record on this file will begin a new physical record. We shall describe two macros, RELSEIN and RELSEOUT, for the two cases.

The RELSEIN macro is similar to the GETBK macro without the final GET:

```
RELSEIN     MACRO     (fcb)
¢L          MOVEH     F
            ONNODE (BB,TRAILER,EOF,ELSE) GOTO (NEXT,¢LL,¢LL,¢L)
            MOVEH     F
            ONNODE (RB,BE,BSN) GOTO (¢LL,¢L,NEXT)
            MOVEH     F
¢LL
RELSEIN     MEND
```

The RELSEOUT macro is essentially the PUTBK macro without the final PUT:

```
RELSEOUT    MACRO     (fcb)
            t ← record size
            record size ← MAXBLOCK
            PUT (fcb)
            PUTSZ (fcb,0)
            record size ← t
RELSEOUT    MEND
```

9.6 Device Positioning Commands

These commands handle the positioning of input/output devices. Some of these commands apply only to unlabeled or multifile tapes. Since we have not attempted to model these, only verbal descriptions will be given.

9.6.1 REWIND

```
REWIND      MACRO    (fcb)
            MOVEH    R
            MOVEH    F
REWIND      MEND
```

9.6.2 WEF (multifile)

The WEF macro writes a file mark or an output file. A file mark is a single character record. If the character is 17_8, it is interpreted as a standard end-of-file. Otherwise it triggers a call to a user provided routine. The calling sequence is:

CALL WEF (fcb, file mark).

9.6.3 FSTFM (multifile,unlabeled)

The FSTFM macro forward spaces an unlabeled multifile tape to a position immediately following the n^{th} succeeding standard end-of-file. The calling sequence is:

CALL FSTFM (fcb,n).

9.6.4 BSTFM (multifile,unlabeled)

The BSTFM macro back spaces an unlabeled multifile tape to a position immediately following the n^{th} preceding standard end-of-file. The calling sequence is:

CALL BSTFM (fcb,n).

9.6.5 FSREC (tape only)

The FSREC macro is used to space over the next n physical records on the designated magnetic tape file in a forward direction. The calling sequence is:

CALL FSREC (fcb,n,eof)

The tape is positioned immediately after the n^{th} physical record which follows the initial position of the tape. For a buffered file, if the last command issued for that file referenced a logical record, then the initial position of the tape is assumed to be immediately after the physical record that contained that logical record. If a file mark (any single character record) is encountered before n physical records have been bypassed, then return is to the location as eof in the calling sequence.

9.6.6 BSREC (tape only)

The BSREC macro is used to space over the n last accessed physical records on the designated magnetic tape in a backward direction. The calling sequence is:

CALL BSREC (fcb, n, eof)

The tape is positioned immediately ahead of the n^{th} physical record which preceded the initial position of the tape. For a buffered file, if the last command issued for that file referenced a logical record, then the initial position of the tape is assumed to be immediately after the physical record that contained that logical record. If a file mark is encountered before n physical records have been bypassed, then return is to the location given as eof in the calling sequence.

9.6.7 FORCE

The FORCE macro is used to force an end-of-reel condition on a magnetic tape file. The calling sequence is:

CALL FORCE (fcb)

If the file is an output file, an end of file (file mark = 17_8) will be written on the current tape. If buffered, the physical record under construction in the buffer will be written prior to writing the file mark. If labeled, the trailer label will be written on the current tape. Unit switching will then be performed. The header label on the new tape will be checked for an expired retention period and the new label, if so indicated, will be written. If the file is an input file, unit switching will be performed. If labeled, the header label on the new tape will be checked.

9.7 Physical Record Processing

GEFRC physical record processing is low level and quite powerful. Input/output is initiated via call to READ or WRITE. This call causes an input/output operation to be started. This operation will occur in parallel with user program execution. Synchronization is obtained by use of a call to WAIT. Input/output may be either consecutive or random, and may include a scatter read or a gather write. These options are specified by the use of a list of data control words (DCWs). It is also possible to specify a "courtesy call" routine to be executed at completion of the input/output request. A program running in courtesy call has certain special properties (for example, it cannot be swapped). Because it is at such a low level, we have not attempted to model GEFRC physical record processing.

9.8 Input/Output Editor Functions

GEFRC includes a set of high level routines specifically designed to provide for certain special purpose requirements of the language processors. These include providing a limited output formatting capability for both printed and punched output. It includes the ability to convert input from COMDEK (compressed)to Hollerith format, to merge an ALTER file with the primary source language input and to create an updated COMDEK output file from the merged input. In addition, the output routines included here provide an accurate interface with the standard output file. We do not attempt to describe these routines in detail but include them here for the sake of completeness.

9.8.1 IOEDIT

The IOEDIT macro initializes the edit functions such as PRINT and PUNCH with parameters which do not vary with each call to these routines. Parameters include heading lines for printed reports, format information for columns 73-80 of punched output, and page numbering information.

9.8.2 RDREC

The RDREC macro obtains the next logical input record from the designated file (with decompression from COMDEK format, if necessary) or from an Alter file of changes to the designated file; and, if required, compresses this logical record into the COMDEK format and insert it into the file designated as K*.

9.8.3 WTREC

The WTREC macro inserts a logical record in the next available position in the designated output file if the record is to be a printed line or a punched card.

9.8.4 PRINT

The PRINT macro inserts a line into the one current printed report whose pages are automatically titled and subtitled, numbered, and controlled by an internal line counter.

9.8.5 EPRINT

The EPRINT macro causes certain special editing of a printed line prior to writing via the PRINT routine.

9.8.6 PUNCH

The PUNCH macro inserts a punched card image in the next available position in the designated output file.

9.9 File Preparation Commands

The file preparation commands are OPEN, CLOSE, SETIN, and SETOUT. These commands prepare a file for proper use by the other commands. We shall describe the macros in English only.

9.9.1 OPEN

The OPEN macro initializes a file so that it may be properly accessed by the other macros. It is implemented by successively refining the general file template to provide information about labels, block serial numbers, blocking, record format, maximum blocksize, and other information needed in the fcb.

9.9.2 CLOSE

The CLOSE macro disconnects a file when no further activity is to be performed on it. It is implemented for output files by writing an end-of-file record, emptying the buffer, and writing a trailer record. The CLOSE is completed by an XFORM which essentially causes the file system to forget the structure of the file.

9.9.3 SETIN

The SETIN macro sets a currently open file to be an input file. This is done by use of XFORM to change the input-output status of the file.

9.9.4 SETOUT

The SETOUT macro sets a currently open file to be an output file. This is done by use of XFORM to change the input-output status of the file.

BIBLIOGRAPHY

Mealy, Cheatham, Farber, Morenoff, and Sattley, Program Transferability Study Group Report, November 1968, NTIS Document #AD-678-589.

Sattley, Millstein, and Warshall, On Program Transferability, November 1970, NTIS Document #AD-716-476.

A Panel Session - Software Transferability, Proceedings SJCC, AFIPS 1969, Vol. 34, AFIPS Press, pp 605-612.

SCOPE Reference Manual 6000 Version 3.3, Control Data 6000 Computer Systems, CDC Pub. No. 6035200, March 1972.

Supervisor and Data Management Services, IBM System/360 Operating System, IBM SRL Form C28-6646-2, November 1968.

Supervisor and Data Management Macro Instructions, IBM System/360 Operating System, IBM SRL Form C28-6647-3.

GE-625/635 File and Record Control, General Electric Information Systems Division, CPB 1003D.

*U.S. GOVERNMENT PRINTING OFFICE: 1972-714-516/9